DOI:10.1145/3316772

Article development led by acmqueue
queue.acm.org

A collaborative approach to reclaiming memory in heterogeneous software systems.

BY ULAN DEGENBAEV, MICHAEL LIPPAUTZ, AND HANNES PAYER

Garbage Collection as a Joint Venture

MANY POPULAR PROGRAMMING languages are executed on top of virtual machines (VMs) that provide critical infrastructure such as automated memory management using garbage collection. Examples include dynamically typed programming languages such as JavaScript and Python, as well as static ones like Java and C#. For such languages the garbage collector periodically traces through objects on the application heap to determine which objects are live and should be kept or dead and can be reclaimed.

The garbage collector is said to manage the application memory, which means the programming language is managed. The main advantage of managed languages is that developers do not have to reason about object lifetimes and free objects manually. Forgetting to free objects leaks memory, and premature freeing results in dangling pointers.

Virtual machines for managed languages may be embedded into larger software systems that are implemented in a different, sometimes unmanaged, programming language, where programmers are responsible for releasing memory that is no longer needed. An example of such a heterogenous software system is Google's Chrome Web browser where the high-performance V8 JavaScript VM (https://v8.dev/) is embedded in the Blink rendering engine that is in charge of rendering a website. Blink renders these pages by interpreting the document object model (DOM; https://www.w3.org/TR/WD-DOM/introduction.html) of a website, which is a cross-platform language-independent representation of the tree structure defined through HTML.

IMAGE BY WACOMKA

Since Blink is written in C++, it implements an abstract DOM representing HTML documents as C++ objects. The DOM C++ objects are wrapped and exposed as objects to JavaScript, which allows scripts to manipulate Web page content directly by modifying the DOM objects. The C++ objects are called *wrappables*, their JavaScript counterparts *wrappers*, and the references connecting these objects *cross-component references*. Even though C++ is an unmanaged language, Blink has its own garbage collector for DOM C++ objects. Cross-component memory management then deals with reclaiming memory in such heterogeneous environments.

V8 and Blink use mark-sweep-compact garbage collectors where a single garbage-collection cycle consists of three phases: *marking*, where live objects are identified; *sweeping*, where dead objects are released; and *compaction*, where live objects are relocated to reduce memory fragmentation. During marking, the garbage collector finds all objects reachable from a defined set of root references, conceptually traversing an object graph, where the nodes of the graph are objects and the edges are fields of objects.

Cross-component references express liveness over component boundaries and have to be modeled explicitly in the graph. The simplest way to manage those references is by treating them as roots into the corresponding component. In other words, references from Blink to V8 would be treated as roots in V8 and vice versa. This creates the problem of reference cycles across components, which is analogous to regular reference cycles[1] within a single garbage-collection system, where objects form groups of strongly connected components that are otherwise unreachable from the live object graph.

Cycles require either manual breaking through the use of weak references or the use of some managed system able to infer liveness by inspecting the system as a whole. Manually breaking a cycle is not always an option because the semantics of the involved objects may require all their referents to stay alive through strong references. Another option would be to restrict the involved components in such a way that cycles cannot be constructed. Note that in the case of Chrome and the Web this is not always possible, as shown later.

While the cycle problem can be avoided by unifying the memory-management systems of two components, it may still be desirable to manage the

memory of the two components independently to preserve separation of concerns, since it is simpler to reuse a component in another system if there are fewer dependencies. For example, V8 is used not only in Chrome, but also in the Node.js server-side runtime, making it undesirable to add Blink-specific knowledge to V8.

Assuming the components cannot be unified, the cross-component reference cycles can lead to either *memory leaks* when graphs involving cycles cannot be reclaimed by the components' garbage collectors, heavily impacting browser performance, or *premature collection of objects* resulting in use-after-free security vulnerabilities and program crashes that put users at risk.

This article describes an approach called cross-component tracing (CCT),[3] which is implemented in V8 and Blink to solve the problem of memory management across component boundaries. Cross-component tracing also integrates nicely with existing tooling infrastructure and improves the debugging capabilities of Chrome DevTools (https://developers.google.com/web/tools/chrome-devtools/).

Cross-component tracing enables efficient, effective, and safe garbage collection across component boundaries.

Separate Worlds for DOM and JavaScript

As mentioned, Chrome encodes the DOM in C++ wrappable objects, and most functionality specified in the HTML standard is provided as C++ code. In contrast, JavaScript is implemented within V8 using a custom object model that is incompatible with C++. When JavaScript application code accesses properties of JavaScript DOM wrapper objects, V8 invokes C++ callbacks in Blink, which make changes to the underlying C++ DOM objects. Conversely, Blink objects can also directly reference JavaScript objects and modify those as needed. For example, Blink can bind fields of JavaScript objects to C++ callbacks that can be used by other JavaScript code.

Both worlds—DOM and JavaScript—are managed by their own trace-based garbage collectors able to reclaim memory that is only transitively rooted within their own heaps. What remains is defining how cross-component references should be treated by these garbage collectors to enable them to effectively collect garbage across components To highlight the problems of leaks and dangling pointers, it is useful to look at a concrete example of JavaScript code and how it can be used to create dynamic content that changes over time.

Figure 1 shows an example that creates a temporary object, a loading bar (`loadingBar`), that is then replaced by actual content (`content`) asynchronously built and swapped in as soon as it is ready. Note that accessing the document element or the body element, or creating the div elements results in pairs of objects in their respective worlds that hold references to each other. While the program itself is written in JavaScript, property look-ups to, for example, the body element and calls to DOM methods `appendChild` and `replaceChild` are forwarded to their corresponding C++ implementations in Blink. Regular JavaScript access, such as setting a parent property, is carried out by V8 on its own objects. It is this seamless integration of JavaScript and the DOM that allows developers to create rich Web applications. At the same time, this concept allows the creation of arbitrary object graphs across component boundaries.

Figure 2 shows a simplified version of the object graph created by the example, where JavaScript objects on the left are connected to their C++ counterparts in the DOM on the right. Java-Script objects, such as the body and div elements, have hardly any references in JavaScript but are mostly used to refer to their corresponding C++ objects. It is thus crucial to define the semantics of cross-component references for the component-local garbage collectors to allow collection of these objects. For example, treating incoming references from Blink into V8 as roots for the V8 garbage collector would always keep the `loadingBar` object alive. Treating such references as uniformly weak would result in reclamation of the body and the div elements by the V8 garbage collector, which would leave behind dangling pointers for Blink.

Besides correctness, another challenge in such an entangled environment is debuggability for developers. While the Web platform allows loose coupling of C++ and JavaScript under

the hood, it is crucial that the APIs for these abstractions are properly encapsulated for Web developers who use HTML and JavaScript, including preventing memory leaks when properly used. To investigate memory leaks in Web pages, developers need tools that allow them to reason seamlessly about the connectivity of objects spanning both V8 and Blink heaps.

Cross-Component Tracing

We propose CCT as a way to tackle the general problem of reference cycles across component boundaries. For CCT, the garbage collectors of all involved components are extended to allow tracing into a different component, managing objects of potentially different programming languages. CCT uses the garbage collector of one component as the *master tracer* to compute the full transitive closure of live objects to break cycles.

Other components assist by providing a *remote tracer* that can traverse the objects of the component when requested by the master tracer. The system can then be treated as one managed heap. As a consequence, the simple algorithm of CCT can be extended to allow moving collectors and incremental or concurrent marking as needed by just following existing garbage collection principles.[8] The pseudocode of the master and remote tracer algorithms is available in our full research article.[3]

For Chrome we developed a version of cross-component tracing where the master tracer for JavaScript objects and the remote tracer for C++ objects are provided by V8 and Blink, respectively. This way V8 can trace through the C++ DOM upon doing a garbage collection, effectively breaking cycles on the V8 and Blink boundary. In this system, Blink garbage collections deal with only the C++ objects and treat the incoming cross-component references from V8 as roots. This way, subsequent invocations of V8's and Blink's garbage collectors can reclaim cycles across the component boundary.

The tracer in V8 makes use of the concept of hidden classes[2] that describe the body of JavaScript objects to find references to other objects, as well as to Blink. The tracer in Blink requires each garbage-collected C++ class to be manually annotated with a method that describes the body of the class, including any references to other managed objects. Since Blink was already garbage-collected before introducing CCT, only minor adjustments to this method were required across the rendering codebase.

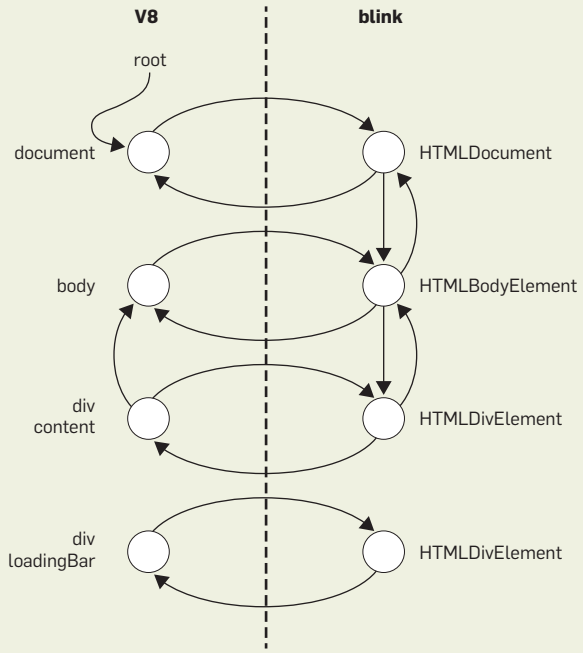
Chrome strives to provide smooth user experiences, updating the screen at 60fps (frames per second), leaving V8 and Blink around 16.6 milliseconds to render a frame. Since marking large heaps may take hundreds of milliseconds, both V8 and Blink employ a technique called *incremental marking*, which means that marking is divided into steps during which objects are marked for only a small amount of time (for example, 1ms).

The application is free to change object references between the steps. This means that the application may hide a reference to an unmarked object in an already-marked object, which would result in premature collection of a live object. Incremental marking requires a garbage collector to keep the marking state consistent by preserving the strong tri-color-marking invariant.[8] This invariant states that fully marked objects are allowed to point only to

Figure 1. JavaScript example interacting with the DOM.

```
<!DOCTYPE html>
<html>
  <body><script>
    function fetchContent(callback) {
      // Emulate network request and content creation.
      setTimeout(callback, 1000);
    }
    function run() {
      const loadingBar = document.createElement("div");
      document.body.appendChild(loadingBar);
      fetchContent(() => {
        const content = document.createElement("div");
        document.body.replaceChild(content, loadingBar);
        content.parent = document.body;
      });
    }
    document.addEventListener("DOMContentLoaded", run);
  </script></body>
</html>
```

Figure 2. Object graph spanning JavaScript and the DOM.



objects that are also fully marked or stashed somewhere for processing. V8 and Blink preserve the marking invariant using a conservative Dijkstra-style write barrier[6] that ensures that writing a value into an object also marks the value. In fact, V8 even provides concurrent marking on a background thread this way while relying on incremental tracing in Blink.[5]

To make this concrete, Figure 3 illustrates CCT where V8 traces and marks objects in JavaScript, as well as C++. Objects transitively reachable by V8's root object are marked black. Subsequently, any unreachable objects (`loadingBar`, in this example) are reclaimed by the garbage collector. Note that from V8's point of view, there is no d ifference between the div elements `content` and `loadingBar`, and only CCT makes it clear which object can be reclaimed by V8's garbage collector. Once the unreachable V8 object is gone, any subsequent garbage collections in Blink will not see a root for the corresponding `HTMLDivElement` and reclaim the other half of the wrapper-wrappable pair.

Figure 3. Cross-component garbage collection.

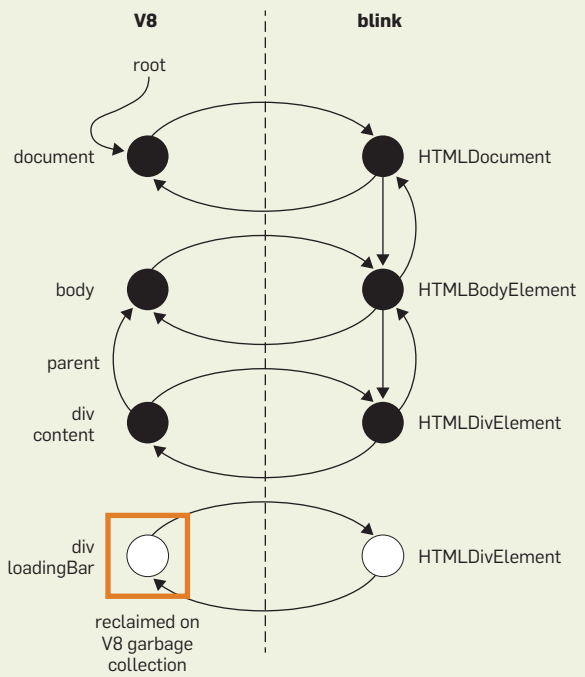


Figure 4. Leaking the callback.

```
function fetchContent(callback) {
      // Emulate network request and content creation.
      setTimeout(callback, 1000);
      fetchContent.internalState = callback;
}
```

Figure 5. Retaining path of the leaking DIV element.

Constructor	Distance
▼Detached HTMLDivElement	5
▶Detached HTMLDivElement @12687 ✂	5
Retainers	
Object	Distance
▼loadingBar in system / Context @52733	4
▼context in () @48313 ☐ example.html:13	3
▼internalState in fetchContent() @35425 ☐ example.html:4	2
▶fetchContent in Window / @4167 ☐	1

In Chrome, CCT replaced its predecessor, called *object grouping*, in version 57. Object grouping was based on over-approximating liveness across component boundaries by keeping all wrappers and wrappables alive in a given DOM tree as long as a single wrapper was held alive through JavaScript. This assumption was reasonable at the time it was implemented, when modification of the DOM from wrappers occurred infrequently. However, the over-approximation had two major shortcomings: It kept more memory alive than needed, which in times of ever-growing Web applications increased already strong memory pressure in the browser; and, the original algorithm was not designed for incremental processing, which, compared with CCT, resulted in longer garbage-collection pause times.

Incremental CCT as implemented today in Chrome eliminates those problems by providing a much better approximation by computing liveness of objects through reachability and by enabling incremental processing. The detailed performance analysis can be found in the main research paper.[3] We are currently working on concurrent marking of the Blink C++ heap and on integrating CCT into such a scheme.

Debugging

Memory-leak bugs are a widespread problem haunting Web applications today.[7] Powerful language constructs such as closures make it easy for a Web developer to accidentally extend the lifetimes of JavaScript and DOM objects, resulting in higher memory usage than necessary. As a concrete example,

let's assume that the `fetchContent` function from Figure 1 keeps, perhaps because of a bug, an internal reference to the provided callback, as shown in Figure 4.

Without knowing the implementation of the `fetchContent` function, a Web developer observes that the `loadingBar` element from the previous example is not reclaimed by the garbage collector. Can debugging tools help track down why the element is leaking?

The tracing infrastructure needed for cross-component garbage collection can be applied to improve memory debugging. Chrome DevTools uses the infrastructure to capture and visualize the object graph spanning JavaScript and DOM objects. The tool allows Web developers to query why a particular object is not reclaimed by the garbage collector. It presents the answer in the form of a *retaining path*, which runs from the object to the garbage-collection root. Figure 5 shows the retaining path for the leaking `loadingBar` element. The path shows that the leaking DOM element is captured by the `loadingBar` variable in the environment (called *context* in V8) of an anonymous closure, which is retained by the `internalState` field of the `fetchContent` function. By inspecting each node of the path, the Web developer can pinpoint the source of the leak. Thanks to the cross-component tracing, the path seamlessly crosses the DOM and JavaScript boundary.[4]

Reclaiming Memory in Other Heterogeneous Systems

Web browsers are particularly interesting systems, as all major browser engines separate DOM and JavaScript objects in a similar way (that is, by providing different heaps for those objects). Similar to Blink and V8, all those browsers encode their DOM in C++ and must rely on a custom object model for JavaScript. All Blink-derived systems (for example, Chrome, Opera, and Electron) rely on CCT to handle cross-component references. The Gecko rendering engine that powers Firefox uses reference counting to manage DOM objects. An additional incremental cycle collector[1] that wakes up periodically ensures that such cycles are eventually collected. WebKit, the engine running inside Safari, uses reference counting for the C++ DOM with an additional system that computes liveness across the wrapper/wrappable boundary in the final pause of a garbage-collection cycle. Unsurprisingly, all major browsers have mechanisms to deal with these kinds of cycles, as memory leaks in longer-running websites would otherwise be inevitable and would observably impact browser performance.

More interestingly, though, we are not aware of other sophisticated systems integrating VMs that provide cross-component memory management. While VMs often provide bridges for integration in other systems, such as Java Native Interface (JNI) and NativeScript, cross-component references require manual management in all of them. Developers using those systems must manually create and destroy links that can form cycles. This is error prone and can lead to the aforementioned problems.

Conclusion

Cross-component tracing is a way to solve the problem of reference cycles across component boundaries. This problem appears as soon as components can form arbitrary object graphs with nontrivial ownership across API boundaries. An incremental version of CCT is implemented in V8 and Blink, enabling effective and efficient reclamation of memory in a safe manner—without introducing dangling pointers that could lead to program crashes or security vulnerabilities in Chrome or Chromium-derived browsers. The same tracing system is reused by Chrome DevTools to visualize retaining paths of objects independent of whether they are managed in C++ or JavaScript.

Note, however, that CCT comes with significant implementation overhead, as it requires implementations of tracers in each component. Ultimately, implementers need to weigh the effort of either avoiding cycles by enforcing restrictions on their systems or implementing a mechanism to reclaim cycles, such as CCT. Chrome was already equipped with garbage collectors in V8 and Blink, and thus we chose to implement a generic solution such as CCT that allows the systems on top to stay as flexible as needed.

CCT is implemented not only in Chrome, but also in other software systems that use V8 and Chrome, such as the popular Opera Web browser and Electron. Cobalt, a high-performance, small-footprint platform providing a subset of HTML5, CSS, and JavaScript used for embedded devices such as TVs, implemented cross-component tracing inspired by our system to manage its memory. C

Related articles on queue.acm.org

Idle-Time Garbage-Collection Scheduling
Ulan Degenbaev et al.
https://queue.acm.org/detail.cfm?id=2977741

Real-time Garbage Collection
David F. Bacon
https://queue.acm.org/detail.cfm?id=1217268

Leaking Space
Neil Mitchell
https://queue.acm.org/detail.cfm?id=2538488

References

1. Bacon, D.F. and Rajan, V.T. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conf. Object-Oriented Programming*. Springer-Verlag, London, U.K., 2001, 207–235; https://doi.org/10.1007/3-540-45337-7_12.
2. Chambers, C., Ungar, D. and Lee, E. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In Proceedings of the Conf. Object-Oriented Programming Systems, Languages and Applications. ACM SIGPLAN, 1989, 49–70; https://dl.acm.org/citation.cfm?doid=74877.74884.
3. Degenbaev, U. et al. Cross-component garbage collection. In Proceedings of the ACM on Programming Languages 2, OOPSLA Article 151, 2018; https://dl.acm.org/citation.cfm?doid=3288538.3276521.
4. Degenbaev, U., Filippov, A., Lippautz, M. and Payer, H. Tracing from JS to the DOM and back again. V8, 2018; https://v8.dev/blog/tracing-js-dom.
5. Degenbaev, U., Lippautz, M. and Payer, H. Concurrent marking in V8. V8, 2018; https://v8.dev/blog/concurrent-marking.
6. Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S. and Steffens, E.F.M. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM 21*, 11 (Nov. 1978), 966–975; https://dl.acm.org/citation.cfm?doid=359642.359655.
7. Hablich, M. and Payer, H. Lessons learned from the memory roadshow; https://bit.ly/2018-memory-roadshow.
8. Jones, R., Hosking, A. and Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall, 2012.

Ulan Degenbaev is a software engineer at Google, working on the garbage collector of the V8 JavaScript engine.

Michael Lippautz is a software engineer at Google, where he works on garbage collection for the V8 JavaScript virtual machine and the Blink rendering engine. Previously, he worked on Google's Dart virtual machine.

Hannes Payer is a software engineer at Google, where he works on the V8 JavaScript virtual machine. Previously, he worked on Google's Dart virtual machine and various Java virtual machines.