Cross-Component Garbage Collection

ULAN DEGENBAEV, Google, Germany
JOCHEN EISINGER, Google, Germany
KENTARO HARA, Google, Japan
MARCEL HLOPKO, Google, Germany
MICHAEL LIPPAUTZ, Google, Germany
HANNES PAYER, Google, Germany

Embedding a modern language runtime as a component in a larger software system is popular these days. Communication between these systems often requires keeping references to each others' objects. In this paper we present and discuss the problem of cross-component memory management where reference cycles across component boundaries may lead to memory leaks and premature reclamation of objects may lead to dangling cross-component references. We provide a generic algorithm for effective, efficient, and safe garbage collection over component boundaries, which we call cross-component tracing. We designed and implemented cross-component tracing in the Chrome web browser where the JavaScript virtual machine V8 is embedded into the rendering engine Blink. Cross-component tracing from V8's JavaScript heap to Blink's C++ heap improves garbage collection latency and eliminates long-standing memory leaks for real websites in Chrome. We show how cross-component tracing can help web developers to reason about reachability and retainment of objects spanning both V8 and Blink components based on Chrome's heap snapshot memory tool. Cross-component tracing was enabled by default for all websites in Chrome version 57 and is also deployed in other widely used software systems such as Opera, Cobalt, and Electron.

CCS Concepts: • **Software and its engineering → Garbage collection**; *Memory management*; *Runtime environments*;

Additional Key Words and Phrases: Garbage Collection, Memory Management, Runtime Environments, Language Implementation

ACM Reference Format:
Ulan Degenbaev, Jochen Eisinger, Kentaro Hara, Marcel Hlopko, Michael Lippautz, and Hannes Payer. 2018. Cross-Component Garbage Collection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 151 (November 2018), 24 pages. https://doi.org/10.1145/3276521

1 INTRODUCTION

Most modern language runtime systems such as the V8 JavaScript virtual machine (VM) manage application memory automatically with garbage collection. The garbage collector periodically traces through objects on the managed heap to determine live objects and reclaim memory. While VMs often manage their own heaps in isolation, they can also be embedded into larger software

Authors' addresses: Ulan Degenbaev, Google, Germany, ulan@google.com; Jochen Eisinger, Google, Germany, eisinger@google.com; Kentaro Hara, Google, Japan , haraken@google.com; Marcel Hlopko, Google, Germany, hlopko@google.com; Michael Lippautz, Google, Germany, mlippautz@google.com; Hannes Payer, Google, Germany, hpayer@google.com.

2475-1421/2018/11-ART151
https://doi.org/10.1145/3276521

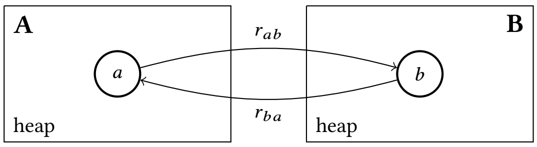


Fig. 1. Reference cycle across component boundaries

systems that are implemented in a different, often unmanaged, programming language. In our case, V8 is embedded in the Blink rendering engine of Chrome which is implemented in C++.

In such heterogeneous software systems, efficient communication is often made possible by allowing objects in one component to store direct pointers to objects in another component, which we call *cross-component references*. An important example is the Blink implementation of the document object model (DOM) representing HTML documents as C++ objects. The DOM C++ objects are *wrapped* and exposed as objects to JavaScript, which allows scripts to manipulate webpage content directly by mutating the objects of the DOM. When JavaScript application code accesses properties on these wrapper objects, V8 invokes embedder (Blink) callbacks which make changes to the underlying C++ DOM objects. Conversely, Blink objects can also directly reference JavaScript objects. These cross-component references express liveness over component boundaries and can create potential memory leaks if cycles are not handled carefully.

Figure 1 shows an example of the cycle problem. Let us assume two objects a and b in their respective components A and B. (For full generality, these single objects could be replaced by arbitrary subgraphs of objects.) Objects in those components can only be live if they are reachable from the root set or through a reference by another live object. If A and B do not share any information on their object graphs, they must conservatively assume that any incoming reference from another component is a root; otherwise an object might be prematurely reclaimed resulting in a dangling pointer and eventually in a program crash or security exploit. Treating references from outside components as roots becomes a problem once a cycle is formed, e.g., through a and b referring to each other through references r_{ab} and r_{ba}, respectively. Cycle creation is unavoidable in the general case as objects might have independent requirements that force them to keep any number of other objects in other components alive. In general, cycles can be arbitrarily large, crossing the component boundary multiple times, and detection may require iterating all objects in A and B in the worst case.

Cross-component cycles exhibit the same problems as reference-counting garbage collectors [Bacon and Rajan 2001; Christopher 1984], where objects form groups of strongly-connected components that are otherwise unreachable from the live object graph. Cycles either require manual breaking through the use of weak references or the use of some managed system that is able to infer liveness by inspecting the system as a whole. Manually breaking a cycle is not always an option as the semantics of the involved objects may require all their referents to stay alive, through strong references.

While the cycle problem can be avoided by unifying the memory management systems of two components, it may still be desirable to manage memory of components independently to preserve separation of concerns, since it may be 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 framework, making it undesirable to add Blink-specific knowledge to V8.

Cross-component memory management is a challenge in several other systems, such as software systems that embed popular scripting languages, in particular all popular web browsers have to solve this problem on some level. We have found that system implementers may not be aware of the cross-component memory management problem or viable solutions to the problem until late in integration. For example, NativeScript [NativeScript 2018] is a popular framework for implementing native platform independent JavaScript applications for mobile devices embedding V8 in the Android ART runtime[1]. The framework authors noticed that managing references across components is complicated and may result in memory leaks or program crashes due to dangling pointers [Slavchev 2014, 2017]. More recently, Cobalt [Cobalt Project Authors 2018] – 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 their memory. Another popular software system that suffered from complicated cross-component memory management issues was the Flash player embedding the ActionScript virtual machine [Mozilla Contributors 2011].

Despite the problem of cross-component references being present in many widely used software systems, it is not discussed in detail in literature to the best of our knowledge. This highlights the importance of our contributions:

(1) We present the cross-component memory management problem.
(2) We provide a generic algorithm that integrates naturally with trace-based garbage collection, which we call *cross-component tracing* (CCT).
(3) We provide a concrete implementation based on Chrome's V8 JavaScript VM and the Blink rendering engine and demonstrate performance improvements over the previously used cross-component memory management system in Chrome.
(4) We show that CCT increases the capabilities of memory tooling by providing precise retainment information across component boundaries and apply it to Chrome's DevTools.

The rest of the paper is structured as follows. In Section 2 we introduce the memory management systems of V8 and Blink. In Section 3 we discuss object grouping, an approach previously used by Chrome to manage cross-component references between the V8 and Blink heaps. We outline limitations of object grouping that may result in memory leaks and long garbage collection pause times. Section 4 introduces CCT, a tracing based cross-component memory management system that provides effective, efficient, and safe garbage collection over component boundaries. We show how CCT improves Chrome's memory tooling. Section 5 discusses related work. We demonstrate in Section 6 that CCT can reduce garbage collection pause times in various real-world web applications, while introducing no significant overhead in total garbage collection time or memory consumption and conclude the paper in Section 7.

2 CHROME MEMORY MANAGEMENT

In this section we give a brief overview of the memory management systems used in Chrome to manage the JavaScript and the DOM memory.

2.1 V8 Garbage Collection

V8 is an industrial-strength open-source virtual machine for JavaScript. Any C++ application can embed V8 using its public API. V8 is embedded into several software systems, including the Blink rendering engine used in both Chrome and the Opera web browser, as well as Node.js, a framework [Tilkov and Vinoski 2010] for server-side computing. JavaScript has only one thread, often called the *main thread*, and each JavaScript context has its own private heap. Hence, allocations in JavaScript require no synchronization and during collection the garbage collector needs to synchronize only with the main thread.

[1] https://source.android.com/devices/tech/dalvik

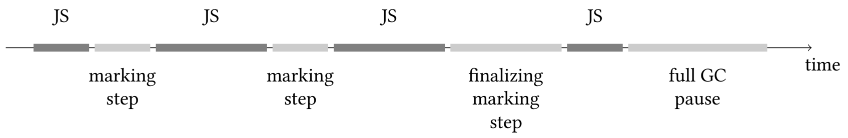


Fig. 2. Incremental marking in V8

V8 uses a generational garbage collection strategy with the heap separated into a small young generation for new objects and a larger old generation for long living objects.

The young generation uses a semi-space strategy where new objects are initially allocated in the young generation's active semi-space using bump-pointer allocation inline in compiled code [Clifford et al. 2014]. A young generation garbage collection is triggered when the active semi-space becomes full and uses a parallel variant [Halstead 1984] of Cheney's algorithm [Cheney 1970] with dynamic work stealing for collection[2]. Objects which have been moved once in the young generation are promoted to the old generation.

The old generation uses bump pointer allocation on the fast path for pretenured allocations [Clifford et al. 2015], also inline in generated code. Free old generation memory blocks are organized in segregated free-lists and are used to refill the current bump-pointer span. A full garbage collection is triggered when the size of live objects exceeds a limit determined by V8's heap growing strategy. A full collection collects both the old and young generation using a mark-sweep-compact strategy with several optimizations to improve latency and memory consumption. V8 marks live objects incrementally in many small steps to avoid long marking pauses, with a target pause time less than 1ms. A Dijkstra-style write barrier [Dijkstra et al. 1978] is used to maintain a tricolor marking scheme. When incremental marking nears completion, finalization marking steps are scheduled to prepare for the full garbage collection pause, i.e. black allocation is enabled to avoid further marking load, weak references are preprocessed, roots are scanned, etc. Figure 2 illustrates the behavior of a full garbage collection, making use of incremental steps before invoking the actual collector. Sweeping is performed concurrently to the main thread and in parallel by multiple threads. After starting the sweeper threads, the young generation is evacuated since marking has already obtained requisite liveness information. Live objects in the young generation are either copied to the other semi-space or promoted to the old generation. Compaction is used to reduce memory fragmentation in the old generation and is done in parallel to reduce the time spent where the main thread is stopped. V8 takes advantage of idle time garbage collection scheduling [Degenbaev et al. 2016] to perform as much garbage collection work during idle time as possible.

2.2 Blink Memory Management

Chrome embeds the Blink rendering engine which is a fork of the WebKit[3] rendering engine and is mostly written in C++. Blink employs a mix of memory management strategies. Memory in Blink is either completely unmanaged, managed via reference counting with manually broken cycles, or fully managed using the Oilpan garbage collector [Ager et al. 2013].

While JavaScript only supports single-threaded execution, the Blink rendering engine is highly multi-threaded. Oilpan provides thread-local heaps with infrastructure to keep thread-local references and cross-thread references to either managed or unmanaged memory.

Oilpan implements a single generation mark-sweep-compact garbage collection strategy for DOM objects and objects closely related to the DOM. Before using Oilpan, these objects were

[2]https://v8.dev/blog/orinoco-parallel-scavenger
[3]https://webkit.org

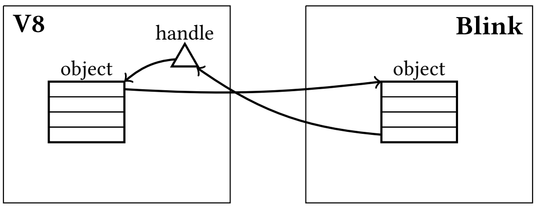


Fig. 3. V8 and Blink: Cross-component references

reference-counted and often subject to memory leaks, one of the main motivations to move to a tracing-based garbage collection system. Allocation is backed by pages that use bump-pointer style allocation when possible and fall back to worst-fit allocation scheme based on free lists. At the start of garbage collection, Oilpan conservatively [Boehm and Weiser 1988] scans the execution stack for pointers into the managed heap. Garbage collection scheduling heuristics are tuned to start most collections at the top level of the browser event loop where the execution stack is empty. If the execution stack is empty, tracing has precise information for stack roots, and objects can be safely moved. Otherwise, compaction is disabled. Oilpan marks the entire heap in a single, potentially large, pause. After marking, Oilpan triggers incremental sweeping and compacts pages containing container backing stores, such as backing stores for vectors and hash maps [Finne 2016]. Sweeping is done either in idle time using idle tasks or lazily upon allocation if there is not enough space in the free list.

While in principle Blink and V8 heaps are independent, in practice, a significant portion of references are cross-component references that connect both systems. A collaboration between both garbage collectors is required for effective reclamation of memory, reducing so-called floating garbage [Jones et al. 2012].

2.3 Bindings: Connecting V8 and Blink using Cross-Component References

So far we considered V8 and Blink as separate components. The *bindings* layer of Blink uses the V8 API to glue together V8 with the rest of Blink. This way Blink exposes some of its *wrappable* C++ objects to JavaScript through *wrapper* JavaScript objects. The interfaces that are exposed are specified in the web interface definition language (WebIDL)[4]. The actual C++ glue code is compiled from the WebIDL description in an automatic fashion. As discussed in other work [Brown et al. 2017] this approach ensures that APIs are used as intended handling return values and error paths.

The bindings layer holds *cross-component references* to JavaScript objects using *handles* exposed by the public API of V8. A handle is essentially an indirection; it is a structure consisting of a single field that stores a pointer to the target object [Craciunas et al. 2008; Kalibera and Jones 2011]. The indirection via a handle is required because V8 can move objects when compacting its heap. The embedder, Blink in this case, can specify a handle to be either strong or weak. The V8 garbage collector treats a strong handle as a root: the object pointed to by the handle is always kept alive until the handle is explicitly destroyed by the embedder. Like weak references, a weak handle alone does not keep the target object alive. A weak handle can have an embedder-supplied finalizer which is invoked by the collector before the object is reclaimed. V8 also provides an API for iterating strong and weak handles.

Cross-component references from V8 to Blink are implemented as direct references from JavaScript objects to Blink objects. There is no need for an indirection through handles as no backing stores (which Oilpan may compact) can be referenced directly. The V8 garbage collector

[4]Blink uses its own dialect of WebIDL as described in https://www.w3.org/TR/WebIDL-1/.

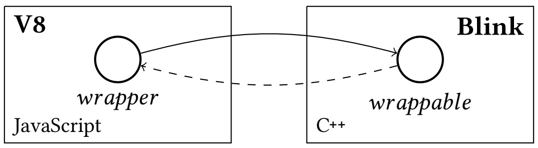


Fig. 4. References between V8 and the Blink renderer

can differentiate between pointers on its managed heap and pointers to the embedder heap by inspecting the type of the object. Additionally, each JavaScript object containing such a reference is also referred to by a V8 handle. Figure 3 depicts such a scenario with cross-component references from V8 to Blink and vice versa. In the following we simplify the object graphs by hiding handles and object layout. On a high level the abstraction through V8's handle system can be seen as a remembered set [Ungar 1984] for maintaining cross-component references. For example, using the remembered set Oilpan can perform local garbage collections by treating incoming cross-component references as roots. Note that the cross-component references from V8 into Oilpan can be found by iterating V8's handles and following the corresponding references back into Oilpan.

Since neither JavaScript nor the WebIDL specification impose any restrictions on DOM composition, the resulting object graphs are fully general, potentially leading to the cycle problem as described in Section 1.

3 OBJECT GROUPING

Recall that cyclic cross-component references from wrapper objects to wrappable objects and vice versa tie their lifetime together: they can only live and die together. These references would lead to memory leaks if either were considered as a root. To avoid this problem, cross-component references from Blink to V8 are marked as weak roots, allowing the wrapper to be collected by V8 if it is otherwise unreachable. In Figure 4, this is denoted by a dashed edge. As V8's garbage collector has no visibility into the references between Blink objects, it might collect a wrapper even though it is still reachable from another wrapper through Blink references between their respective wrappables. For an example see Figure 5 where b is reachable through the transition $a \longrightarrow a' \longrightarrow b' \longrightarrow b$.

The majority of cross-component references from V8 to Blink are references connecting JavaScript objects to C++ objects representing nodes in the DOM. Furthermore, edges in the DOM are bidirectional, which means any live node in the DOM tree will keep the whole DOM tree alive. This bit of domain-specific knowledge makes it possible to group JavaScript wrapper objects by their corresponding DOM trees, a technique called *object grouping* which was used in Chrome until version 57. Since all lifetimes of a group must be the same, a group can either be a *live group*, if it contains at least one live wrapper, or a *dead group*, otherwise. Once a group is dead it is collected by V8. A program can also mutate the DOM tree, which results in DOM nodes moving between groups, effectively invalidating already collected liveness information.

An example of object grouping is illustrated in Figure 5. The fictional DOM trees are g_1 containing the DOM nodes a' and b', and g_2 containing the DOM node c'. Then the wrapper objects a and b are grouped together in g_1 using object grouping as their corresponding wrappables are contained within the same DOM tree. Similarly, wrapper c forms its own group g_2 since its wrappable c' is the only node in the DOM tree. Now, if o happens to be considered live by V8, then g_1 is also considered live because of the reference between o and a. Wrapper b will be kept alive even though there is no connection on the V8 heap between a and b. In contrast, g_2 and its objects – only c in this example – will be collected upon a garbage collection if c is not itself in V8's root set of objects.

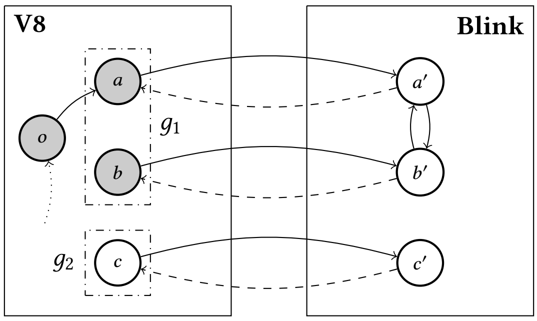


Fig. 5. Object grouping

Additional relationships between JavaScript objects due to references in Blink must be explicitly modeled by introducing temporary JavaScript references between the involved objects. The references are only temporary as there is no mechanism for tracking changes in the DOM object graph and updating the references accordingly.

One last complication is that Blink can also keep references to JavaScript objects that aren't wrappers. If a cycle exists in the system that involves such references, it will not be automatically detected by the object grouping algorithm but instead will result in a memory leak. An example is Blink objects keeping references to callbacks in JavaScript[5].

3.1 Limitations of Object Grouping

Object grouping has two major shortcomings which are (1) latency and (2) the complexity of assigning object groups in environments where the fundamental assumption of a fully connected graph does not hold, often resulting in premature collection of objects or memory leaks.

Performance. Object grouping has two primary performance problems. The first problem is that establishing liveness of groups requires considering all objects, even those that are only part of dead groups. Figure 6 illustrates that problem. In this example, even though all objects in V8 are dead, object grouping still requires considering all cross-component references from Blink to V8 to build up the groups g_1 and g_2. This issue cannot be mitigated as the key in object grouping is assigning each object, dead or alive, to a group. Note that this is not the same as holding a subtree live through a single reference that gets discovered in a tracing garbage collector, as this problem still exists on top of creating groups for dead objects. The second problem is that object grouping information is only correct as long as JavaScript is not mutating the DOM graph. Correctness is achieved when groups are created and processed during the finalization pause of the garbage collector. Creating and establishing liveness of the groups is bounded in the number of cross-component references, i.e., the total number of nodes in all DOM trees. Additionally, entries into other subgraphs of the JavaScript heap are only discovered once groups have been fully processed. To mitigate this issue, once a heuristic determines that the heap is close to being fully marked, object grouping is repeatedly performed between incremental marking steps.

Code Complexity. Object grouping was designed around the property of fully-connected groups that live and die together. Graphs that involve keeping references from DOM objects that aren't part of a DOM tree to JavaScript require temporary references to be correctly represented. In practice, the majority of objects are part of the DOM tree. However, for references between DOM objects

[5]https://crbug.com/501866

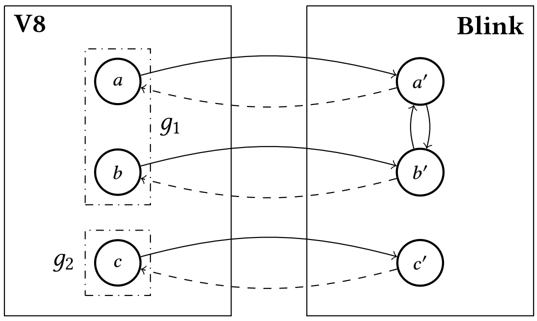


Fig. 6. Limitations of object grouping: Considering dead objects and groups

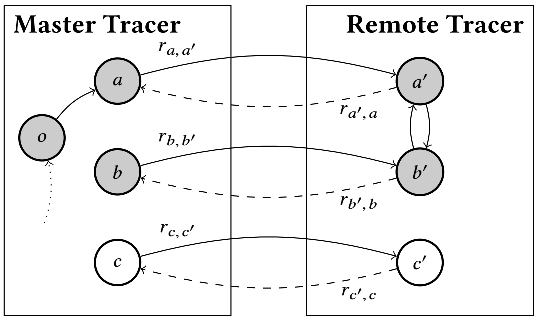


Fig. 7. Cross-component tracing

that are not part of the DOM tree, and references to JavaScript objects that are not a wrapper, the garbage collector must be able to fully describe the entire system at any point in time. Maintaining this global knowledge is not feasible in practice, as manifested in several Blink bugs where wrappers were either collected prematurely[6], or memory leaked[7].

4 CROSS-COMPONENT TRACING

In this section we first discuss an abstract setting of components and an algorithm with building blocks for computing precise liveness of objects across components. The generic algorithm can be used as a sketch for other systems that require cross-component memory management. We conclude this section by relating those general concepts back to V8 and Blink. More precisely we show how CCT is implemented in V8 and Blink and how it improves Chrome's memory tooling.

Similar to solving the problem of reference cycles in garbage collected systems, the algorithm relies on tracing to compute liveness, correctly handling reference cycles across component boundaries. We assume that the component that triggers collection is managed by a garbage collector and refer to its tracing infrastructure as *master tracer*. We refer to the system that assists this garbage collector in another component as *remote tracer*. The remote tracer must at least have a mechanism to trace its heap, either because it too is garbage collected or a tracing mechanism can be built.

Cross-component tracing can be seen as tracing that ignores component boundaries. Figure 7 illustrates this by using the same example as introduced for object grouping in Figure 5 with the difference that all references are used for tracing instead of object grouping. During tracing, the

[6]https://crbug.com/329103
[7]https://crbug.com/501866

master tracer will eventually find object a through the other live object o. It then announces $r_{a,a'}$ to the remote tracer, which uses tracing to find all transitively reachable objects a' and b', with its references $r_{a',a}$ and $r_{b',b}$ which are passed back to the master tracer. The master tracer continues tracing from $r_{a',a}$ and $r_{b',b}$ and finds a new live object in b. The master tracer announces reference $r_{b,b'}$ to the remote tracer which can ignore it as it has already been visited. Since there are no other references to trace, tracing is now finished. Note that since there are no objects pointing to either c or c' other than themselves, c is discovered as being dead and collected. c' may be still reachable through some unrelated references in the remote heap.

4.1 Generic Algorithm

The complete generic algorithm is depicted in Figure 8. The algorithm shows cross-component tracing across two components where a garbage collection is initiated by a master tracer (`MasterTracer`; line 1) and assisted by a remote tracer (`RemoteTracer`; line 32). The master tracer and remote tracer mark only objects on their own heaps. The algorithm can be extended to several remote tracers that all handle their own corresponding heaps (see end of this section). We refer to cross-component references through the `is_remote()` predicate. Any object for which `is_remote()` holds is in fact a reference on another heap.

Both the master and the remote tracer maintain their own local marking work lists (`MarkingWorkList`; lines 2 and 33), a data structure to store objects that were discovered during tracing and require tracing of their fields. At the start of the marking phase (`mark_live_objects()`; line 14) the master tracer marks its own roots (line 15) and initiates marking the roots on the remote tracer (line 16), effectively filling both marking work lists with initial objects. It is sufficient to consider only roots in the remote heap that can transitively reach objects which have cross-component references. Note that this set of roots can be a subset of the actual roots. The consequences of the different root sets are discussed after the algorithm description. The marking phase (lines 17-19) then alternates tracing through the master and remote heap. Advancing marking for the master tracer drains its marking work list (lines 22-29) with an algorithm that gets an object, iterates its internal references, and handles them as follows. If the reference is a cross-component reference (line 26), i.e. the `is_remote()` predicate is true, it is announced to the remote tracer (line 27). Otherwise, if the object was not yet marked, it is marked and put on the marking work list (line 29). The remote tracer's marking method (lines 45-53) works exactly the same. `advance_marking(deadline)` takes as progress condition a deadline that can be e.g. real time, number of bytes, or number of objects. `mark_live_objects()` advances marking with `inf` (i.e. infinity) meaning that it marks until the given work list is empty. Both tracers exchange references to objects via their respective work lists, and tracing continues until no more unprocessed objects are available on the marking work lists.

As noted above, it is possible to scan only the roots necessary for cross-component tracing or the full root set. The algorithm guarantees that unmarked objects on the master's heap are unreachable and can be reclaimed. Unmarked objects on the remote heap are only unreachable if the full remote root set was scanned.

As an optimization, the described cross-component garbage collection can be supplemented with faster local garbage collections. During a local garbage collection each component conservatively treats the references from the other component as root references, which may introduce reference cycles and memory leaks. Performing a cross-component garbage collection once in a while is sufficient to break the cycles.

The following extensions are possible but have been omitted from the text for clarity.

```
MasterTracer {
  MarkingWorkList work_list;

  mark_roots():
    // Root marking logic.

  mark(object):
    // Marks the object and returns true on
    // success and false otherwise.

  mark_and_put(object):
    if mark(object): work_list.put(object)

  mark_live_objects():
    mark_roots()
    RemoteTracer.mark_cct_roots()
    while !work_list.empty():
      advance_marking(inf)
      RemoteTracer.advance_marking(inf)

  advance_marking(deadline):
    while !work_list.empty() &&
          !deadline_reached(deadline):
      obj = work_list.get()
      for child in obj:
        if is_remote(child):
          RemoteTracer.mark_and_put(child)
        else:
          mark_and_put(child)
}

RemoteTracer {
  MarkingWorkList work_list;

  mark_cct_roots():
    // Mark roots for CCT.

  mark(object):
    // Marks the object and returns true on
    // success and false otherwise.

  mark_and_put(object):
    if mark(object): work_list.put(object)

  advance_marking(deadline):
    while !work_list.empty() &&
          !deadline_reached(deadline):
      obj = work_list.get()
      for child in obj:
          if is_remote(child):
            MasterTracer.mark_and_put(child)
          else:
            mark_and_put(child)
}
```

Fig. 8. Generic cross-component tracing algorithm

Unlimited Number of Components. The algorithm can be extended to support unlimited number of components. The main differences are: remote objects need to be communicated to their corresponding tracer; and termination needs to consider a fix point across all components where objects are processed until all marking work lists are empty.

```
 1 // Incremental extension.
 2 MasterTracer {
 3   start():
 4     mark_roots()
 5     RemoteTracer.mark_roots()
 6
 7   incremental_step(deadline):
 8     advance_marking(deadline)
 9     RemoteTracer.advance_marking(deadline)
10 }
11
12 // Concurrent extension.
13 MasterTracer/RemoteTracer {
14   start():
15     mark_roots()
16     start_thread(run)
17
18   run():
19     while !wait_for_new_objects_or_terminate():
20       advance_marking(inf)
21 }
```

Fig. 9. Cross-component tracing extensions

Moving Collectors. The generic algorithm relies on references pointing directly to objects. Supporting components that rely on moving garbage collectors, e.g. for compaction, is possible by referring to objects through an indirection such as handles.

Incremental Marking. The extension to support incremental marking is depicted in Figure 9. Marking is started (line 3) by marking the roots in both heaps. Note that root marking in the remote heap is optional at start time if it is performed later when incremental marking finalizes. Afterwards both heaps are processed in alternating order with given deadlines (line 7) interleaved with regular program execution. General infrastructure to support incremental marking, e.g., write barriers and stepwise tracing logic of objects, is also required. The algorithm is opportunistic, i.e., any unprocessed object left on the marking work lists while marking incrementally will be processed during the finalization of this garbage collection phase which has to invoke `mark_live_objects()`.

Concurrent Marking. The extension to support concurrent marking is depicted in Figure 9. Similar to incremental marking, the start of concurrent marking needs to mark roots (line 14) (at least for the master heap). Additionally, threads are started that trace and mark the corresponding heap on another thread. The runner includes a barrier to wait for new objects (line 19) that bails out when all tracers, remote and master, are done with processing their objects. Similarly to incremental marking, general infrastructure to support concurrent marking is required and marking is performed opportunistic.

4.2 V8 & Blink: Implementation of Incremental Cross-Component Tracing

We now provide an overview of how the generic algorithm with the incremental marking and moving collectors extensions map onto V8 and Blink as described in Section 2 and give an overview of various implementation-specific details.

V8 serves as master tracer initiating the garbage collection based on heap-sizing heuristics [Degenbaev et al. 2016; White et al. 2013] that consider the current size of the V8 and Blink managed heaps. The generic methods `advance_marking` and `incremental_step(deadline)` directly map to the existing incremental marking code in V8. V8's `mark_roots` considers all references from

```
DEFINE_CROSS_COMPONENT_TRACING(typeof($a'$)) {
   visitor->traceWrappers($r_{a',b}$);
   visitor->traceWrappers($r_{a',a}$);
}
DEFINE_CROSS_COMPONENT_TRACING(typeof($b'$)) {
   visitor->traceWrappers($r_{b',a}$);
   visitor->traceWrappers($r_{b',b}$);
}
DEFINE_CROSS_COMPONENT_TRACING(typeof($c'$)) {
   visitor->traceWrappers($r_{c',c}$);
}
```

Fig. 10. Explicit tracing definitions

the native execution stack as well as internal and external (API) global references as roots. The knowledge of which references are cross-component references is present in the *hidden class* of an object which describes its shape [Chambers et al. 1989]. The same hidden class is used by V8's garbage collector to determine which slots in an object are JavaScript references.

Blink already ships with its own garbage collector Oilpan. Cross-component tracing reuses some Oilpan infrastructure but all data structures are kept separate. As a result Oilpan garbage collections may interleave with incremental cross-component tracing. An example for shared infrastructure is the location of mark bits which is fixed for each object. Cross-component tracing adds one additional bit in that existing space. In contrast to an explicit tri-color abstraction for the marking state only the black and white colors are encoded in the object header. The gray color indicates that the object is in the marking work list and it is useful for recovering from marking work list overflow. Cross-component tracing does not limit the size of the marking work list, so it does not need the gray color.

Oilpan is currently implemented as stop-the-world garbage collector which is incompatible with the incremental extension of cross-component tracing as it lacks infrastructure to keep the marking state consistent during application execution. Similar to V8, a Dijkstra-style write barrier [Dijkstra et al. 1978] was added to cross-component tracing in Blink to keep the marking state consistent. This barrier is only active when tracing across component and not used during regular Oilpan garbage collections.

In Blink only objects related to the DOM can have cross-component references to V8. This set of objects is significantly smaller than the whole Blink heap. Additionally, Blink guarantees that the native execution stack does not contain references that exclusively retain such DOM objects, i.e., live objects with cross-component references are always reachable through the heap. This is possible because Blink enforces connecting those objects to the heap upon construction through a mix of best practices and debug assertions that fail upon violating this invariant. For cross-component tracing `mark_cct_roots` thus only considers global variables that can reference objects related to the DOM. As discussed in Section 4.1, therefore only objects on the V8 heap can be reclaimed since not all roots of the Blink heap were scanned. This is a trade-off between restricting usage of C++ and increasing complexity of the garbage collection implementation.

Since the object layout of C++ objects is known at compile time, there is no need for implementing a dynamic lookup mechanism for tracing a specific object. Instead, classes are annotated in source code with definitions on how such objects should be traced. Figure 10 illustrates the definitions required for the example in Figure 7, c.f. [Bartlett 1989]. Explicitly specifying which references need to be traced is beneficial as it allows the implementation to only consider subgraphs of the heap managed by Oilpan that contain objects that might contain references to V8 objects.

```html
<html>
  <body onload="run()">
    <div>
      <b id="msg">Hello!</b>
    </div>
  </body>
</html>
```

(a) HTML source

```javascript
class Leak{};
function run() {
  let leak = new Leak();
  function listener() { console.log(leak); }
  let node = document.getElementById("msg");
  node.addEventListener("debug", listener);
}
```

(b) JavaScript source

Fig. 11. A typical memory leak that happens when the web developer forgets to unregister an event listener. All objects in the environment of the listener closure cannot be garbage collected.

The `new_object` methods on both V8 and Blink are implemented on top of buffers to avoid crossing the component boundary for single references. Moreover, all Blink to V8 references are implemented on top of handles to allow V8 to compact its heap (essentially moving objects) which as described before can be seen as remembered set.

Safety. Safety is ensured by two properties: (a) tracers visiting all references that can transitively reach objects with cross-component references; and (b) write barriers ensuring that newly added references during execution are also visited.

For (a) if tracers, master and remote, visit all objects reachable from both root sets then all cross-component references are reached per definition and no objects can be prematurely collected. In our implementation of the master tracer in V8 this holds. For our implementation in Blink we optimize the time spent in tracing by reducing the number of objects that need to be visited. We statically infer through C++ type information which objects can reach objects with cross-component references and only consider those objects during visitation and root scanning, as described in Section 4.2.

For (b), a fundamental problem with incremental and concurrent garbage collection systems is making sure that write barriers are never skipped accidentally. In case of a Dijkstra write barrier this means executing the barrier right after an object has been inserted into the object graph through a write to an existing object. This can be achieved with compiler support by emitting the write barrier code after each write. However, adding the write barrier in an existing C++ application without changing compilers is difficult as every write needs to be intercepted, whether it is explicit through assignment or implicit through copy or move construction within container data structures. We propose a solution to this problem that is correct by construction by providing GC-aware smart pointers to users of DOM code in Blink. Similar to existing smart pointers in the C++ standard library those pointers are used to intercept writes to an object. More specifically, the smart pointer overrides assignment, copy construction, and move construction to invoke the write barrier when needed. Reads are not intercepted. As long as object references are encapsulated in such a smart pointer, the graph is held in a consistent state through incremental cross-component tracing. We enforce the use of smart pointers using the C++ type system and only accepting managed objects as parameters on the internal tracing methods.

4.3 Debugging and Tooling

So far we discussed memory leaks caused by reference cycles across component boundaries. As we have shown, browser developers can fix such leaks by implementing CCT. There is another class of memory leaks which browser developers cannot control – memory leaks on the JavaScript application level. Figure 11 shows an example of JavaScript level memory leak, where an event listener closure captures an object in its environment. The DOM specification requires JavaScript

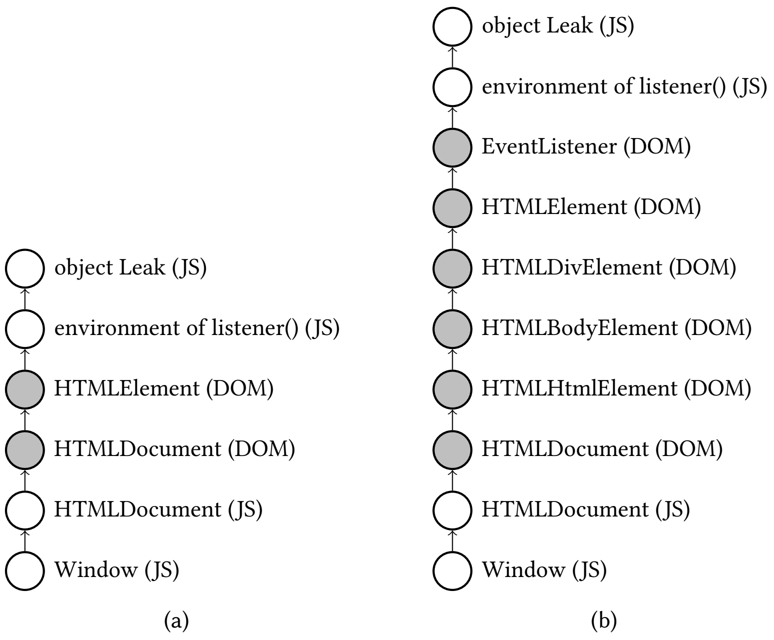


Fig. 12. Retaining path of the leaked object in Chrome DevTools. Path (a) is an approximation based on object grouping. Path (b) is the precise retaining path based on CCT. The JavaScript objects are shown as white nodes. DOM object are shown as gray nodes.

developers to manually remove unused event listeners. Forgetting to remove event listeners is a common source of memory leaks. In general, bugs in JavaScript code can result in leaks of JavaScript and DOM objects. To the garbage collector these leaking objects are indistinguishable from live objects. Browser developers cannot fix these leaks but they can provide tools that help JavaScript developers to find and debug them. Chrome provides a heap snapshot tool as part of DevTools[8] which captures and visualizes the object graph of the JavaScript heap and parts of the Blink heap. Using DevTools, a JavaScript developer can reason why a particular object is not garbage collected by inspecting the retaining path of that object.

The heap snapshot tool captures JavaScript objects and references between objects precisely but until CCT it had only rudimentary support for handling the Blink heap. The tool relied on object grouping and on heuristics to capture DOM objects, which resulted in capturing a subset of the objects and approximating references between them. Lets consider the example in Figure 11. Figure 12 compares the retaining paths using object grouping and using CCT for the leaked object. The EventListener object does not appear in the path shown in Figure 12a, and the path goes from `HTMLDocument` directly to `HTMLElement` bypassing all intermediate nodes because all objects in the same object group are considered connected with each other. For CCT the path in Figure 12b is precise and more actionable because it shows where exactly the event listener is attached in the DOM tree. Precise capturing of DOM objects was implemented by specializing the generic C++ visitation mechanism of CCT. To the best of our knowledge, Chrome DevTools is the only system that can visualize precise object graphs spanning JavaScript and C++ objects. Besides debugging

[8]https://developer.chrome.com/devtools

memory leaks the precise retainment information across components allows developers to reason about general memory usage of their system.

Recent studies have shown that the number one reason for Chrome to crash is webpages running out of memory [Hablich and Payer 2018]. Hence, precise memory tooling is becoming increasingly important. Others have shown how to debug JavaScript memory leaks based on heap snapshots where webpage developers provide scripts that exercise interesting application scenarios [Vilk and Berger 2018]. CCT may improve their work by precisely capturing the DOM object graph.

4.4 Trade-offs

So far we have proposed a novel design that enables effective and efficient reclamation of garbage across component boundaries. We now discuss the trade-offs we have chosen and potential improvements.

Compared to object grouping, we see numerous benefits of CCT. Having a clear interface defining explicitly the dependencies between single objects offers a clean separation of components. CCT only requires a view of each single object through tracing which is a local property, whereas object grouping required a global view of the system to avoid leaks through cycles between groups. CCT also allows for incremental garbage collection without considering the global state, which object grouping requires by iterating all cross-component references. This advantage is clear in the performance results shown in Section 6.

The key for correctness is enforcing the use of of GC-aware smart pointers through the C++ type system. The current design also allows for a future implementation improvement by implementing the concurrent cross-component tracing extension. Even though DOM mutation is specified to be entirely single-threaded, marking can be performed concurrently without interfering with the main thread. Atomic accessors can be encapsulated in the smart pointers, while the barrier ensures consistency of marking information.

5 RELATED WORK

In this section we outline how cross-component references are managed in other software systems. We start out with other web browsers and renderers that similarly to Chrome face the challenge of managing cross-component references between JavaScript and the DOM heap. Afterwards we provide an overview of systems that embed virtual machines and allow references to and from the managed heap. We then take a look into garbage collection in general as well as for C++ specifically and distributed garbage collection systems.

5.1 Other Web Browsers and Renderers

Other popular web browsers implement the setup of the DOM and JavaScript similarly to Chrome and contain a JavaScript virtual machine which allocates JavaScript objects on a managed heap. In each of these systems the DOM is implemented using regular C++ which allocates objects on a separate C++ heap. We did not find any documentation about how Microsoft's Internet Explorer manages cross-component references. The Opera web browser and Electron, a framework for building cross-platform Desktop apps, use Blink and V8 and therefore use incremental cross component garbage collection with their application-specific modifications.

The Gecko rendering engine powers Firefox and uses reference counting to manage C++ memory. To collect cycles that might come either from cyclic data structures such as the DOM tree or references from JavaScript, Gecko implements a cycle collector [Mozilla Contributors 2017]. The implementation is based on work by Bacon and Rajan [2001] without implementing concurrent collection. On a high level, the cycle collector maintains a set of potentially dead objects and regularly detects cycles within this set that then can be deleted. Objects are added to this set whenever their reference count decreases and removed if it increases. To maintain short pause times, the cycle detection itself can be executed incrementally. Various optimizations are implemented to keep the size of the set of potentially dead objects small, e.g., liveness information is propagated from the JavaScript garbage collector, and knowledge about the DOM tree structure is used to add or remove all nodes belonging to a given DOM tree.

The WebKit[9] rendering engine powers Safari and uses reference counting to manage C++ memory. JavaScript wrappers maintain a reference count to their wrapped C++ objects, and the JavaScript engine's garbage collector Riptide [Pizlo 2017] is responsible for maintaining these reference counts. The relationships between C++ objects are expressed using constraints, and Riptide computes a fix-point of those constraints while marking objects. While Riptide itself is concurrent and incremental, the fix-point of the constraints expressing relationships between C++ objects and their wrappers is computed in the atomic pause of the garbage collection cycle.

Additionally to browsers there also exist other HTML renderer implementations such as e.g. Cobalt [Cobalt Project Authors 2018]. Cobalt aims at providing a minimal HTML implementation to run web applications in embedded environments such as TVs. The most recent version of Cobalt uses V8 with its cross-component tracing API without the incremental marking extension and its own DOM implementation.

5.2 Language Interoperability

While virtual machines can run in isolation they are often integrated into larger software systems to provide additional capabilities. Communication between those systems requires holding references of some sort across component boundaries. We are not aware of any other work that allows tracing over component boundaries. The existing solutions default to explicit manual memory management on the component boundaries.

NativeScript [Slavchev 2017] is a popular framework using JavaScript to provide native applications for mobile devices embedding V8 in the Android ART runtime. JavaScript objects may reference Java objects and vice-versa. Reference cycles are broken manually which is error-prone and may lead to either memory leaks or dangling pointers.

JavaConnect [Brichau and Roover 2009] and JNIPort [Geidel 2018] integrate a Java VM into a Smalltalk VM using their respective foreign function interfaces. Cross-component references are roots upon construction and must be explicitly destroyed from Smalltalk using finalizers.

In general, virtual machines provide foreign function interfaces (FFIs) to hold references into other components. For example, for Java, Java Native Interfaces [Oracle Corporation 2018] and JavaConnect [Brichau and Roover 2009] as well as JNIPort [Geidel 2018], allow embedding a JVM into C/C++ and Smalltalk, respectively. These libraries require explicit management of references though.

5.3 Garbage Collection

The idea of dividing managed heaps into different sub parts that can be traced individually is present in other work with a pure focus on improving temporal performance by providing better locality when processing objects [Chicha and Watt 2006; Oancea et al. 2009; Shuf et al. 2002]. Unlike CCT, none of these systems aims at providing safe memory management across component boundaries.

Garbage collection of the DOM requires handling C++ objects. As of today, there exist many different garbage collectors for C++ [Boehm and Weiser 1988; Detlefs 1990, 1992; Edelson 1992]. These collectors position themselves in the garbage collection trade-off space by either being

[9]https://webkit.org

reference counted, trace based, or some hybrid. Furthermore, the collectors are either conservative or precise [Jones et al. 2012].

CCT as presented in this paper relies on tracing and precise stack scanning for JavaScript objects. For C++ the system relies on the fact that garbage collections are performed at the top of the event loop where the native execution stack is empty.

CCT in Blink makes use of custom types that developers need to inherit from and extend for tracing. As others have already noted [Boehm et al. 1991], this avoids wrongly classifying memory as pointers when tracing through object payloads.

Our incremental extension of CCT uses a Dijkstra-style write barrier [Dijkstra et al. 1978].

5.4 Distributed Tracing Garbage Collection

On a high level cross-component memory management is related to distributed garbage collection. However, the environments, i.e., the problem space and their constraints, differ in subtle ways as discussed below.

One of the main differences to traditional distributed garbage collection is that cross-component memory management as described in this paper handles components within a single process, avoiding the problem of network latency and reliability. Additionally, we did not impose any requirements on fault tolerance on the involved components.

Similar to distributed garbage collection [Plainfossé and Shapiro 1995], CCT also requires to synchronize on a global marking state to ensure consistency when reclaiming objects. However, CCT as presented in this paper differs from existing approaches by avoiding complex consistency protocols [Hughes 1985; Ladin and Liskov 1992] or pointer tracking methods [Hudson et al. 1997] that allow for precise reclamation of objects on their local heaps, independently of the global state. Instead, we rely on the fact of having all cross-component references present in a remembered set representation that can be shared in address space among all components. Additionally, we avoid complex synchronization mechanisms by relying on a marking scheme that just moves forward, i.e., objects that were once considered as live stay live during the current garbage collection cycle.

Similar to distributed garbage collected system, local garbage collections for individual heaps are still possible with the presented approach. In this case a separate set of mark bits is built up in isolation and any incoming cross-component references are considered as roots.

6 EXPERIMENTS

We compare four different cross-component memory management systems in Chrome to understand their impact: incremental object grouping (OGI), non-incremental object grouping (OGN), incremental cross-component tracing (CCTI), non-incremental cross-component tracing (CCTN). The non-incremental configurations perform cross-component garbage collection work only during finalization of V8 garbage collection. The OGI configuration builds object groups twice: once in an incremental marking step and again in the finalization phase. Construction of object groups in the incremental step is triggered when the marking work list becomes empty. Newly discovered unmarked objects that belong to a live object group are then added to the marking work list and incremental marking continues. Finalization of garbage collection starts when the marking work list becomes empty again. CCTN performs tracing through Blink only in the finalization phase and CCTI interleaves incremental Blink tracing steps with incremental marking steps of the V8 heap.

We consider the OGI configuration as a baseline since it was the default configuration in Chrome before cross-component tracing. OGN was used in Chrome until early 2015[10]. Starting from Chrome version 57, the CCTI configuration is the new default configuration.

[10] https://codereview.chromium.org/1012593004

6.1 Setup

We use Chrome version 57.0.2987.0 with a local patch that adds run-time flags for switching between browser configurations and fixes to reduce variability in garbage collection scheduling. We run the benchmarks on a Linux workstation with two Intel Xeon E5-2690 V3 12-core 2.60 GHz CPUs and 64GB of main memory.

6.2 Benchmarks

We evaluate each browser configuration on a set of popular webpages using Chrome's Telemetry performance testing framework [Google 2018]. Telemetry runs scripted webpage interactions in Chrome and records all server responses in order to enable predictable replay of these interactions in our test environment, effectively eliminating all unpredictable network traffic. We use official real-world website benchmark sets of Chrome [V8 Project Authors 2017] that mimic common user interactions like infinite scrolling, social network browsing, news webpage browsing, media webpage browsing etc.

The used workloads with URLs and their workload definition can be found in the Chrome's Telemetry repository[11]. Overall we run the following 16 workloads and webpage combinations: *media:imgur*, *media:pinterest*, *media:tumblr*, *media:youtube*, *news:cnn*, *news:flipboard*, *news:hackernews*, *news:nytimes*, *news:reddit*, *news:google*, *news:twitter*, *scroll:discourse*, *scroll:facebook*, *scroll:flickr*, *scroll:tumblr*, and *scroll:twitter*.

6.3 Metrics

We are interested in measuring garbage collection latency, garbage collection throughput, and browser memory usage. We have two metrics for latency: the average pause time of garbage collection finalization and the minimum mutator utilization (MMU) [Cheng and Blelloch 2001]. Finalization is the longest phase in a garbage collection cycle. This is because we can arbitrarily decrease the duration of an incremental marking step by controlling the amount of work performed in the step, but we have no control over the duration of finalization.

The MMU metric measures the impact of multiple consecutive garbage collection steps on the application. It is computed by sliding a time window of fixed size over the execution trace and finding the worst-case window where the most of the time is taken up by garbage collection. The fraction of the time left to the application in this worst-case window is the value of the MMU. Thus, the MMU equal to 1 is the best possible result and the MMU equal to 0 is the worst possible result.

We measured the MMU for window sizes from 1ms to 1000ms and have chosen to present 16.67ms and 50ms time window sizes as they represent the duration of one frame in a 60 frame-per-second animation and the suggested response time for Chrome's RAIL model[12], respectively.

We estimate garbage collection throughput using the total garbage collection time which includes incremental steps and the finalization phase.

By running each webpage 10 times for each configuration we obtain 10 samples per metric. We consider the OGI configuration as the baseline and compare other configurations to the OGI baseline using the Wilcoxon rank sum test [Hollander et al. 2013]. We report the results with 95% confidence intervals.

6.4 Results

Figure 13 shows how garbage collection finalization pause times compare to the OGI baseline. The pause times are normalized relative to the OGI baseline. A confidence interval that includes the

[11]Telemetry: https://chromium.googlesource.com/chromium/src/+/master/tools/perf/benchmarks
[12]RAIL performance model: https://developers.google.com/web/fundamentals/performance/rail

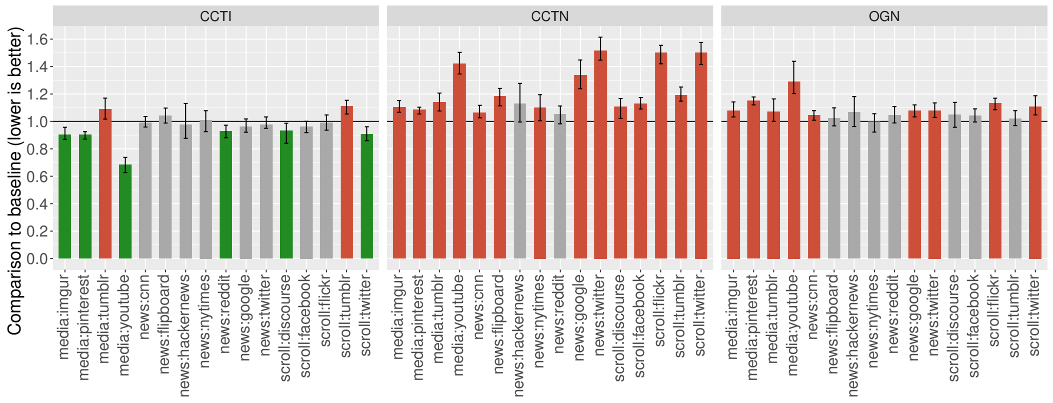


Fig. 13. Finalization pause of garbage collection compared to the OGI baseline

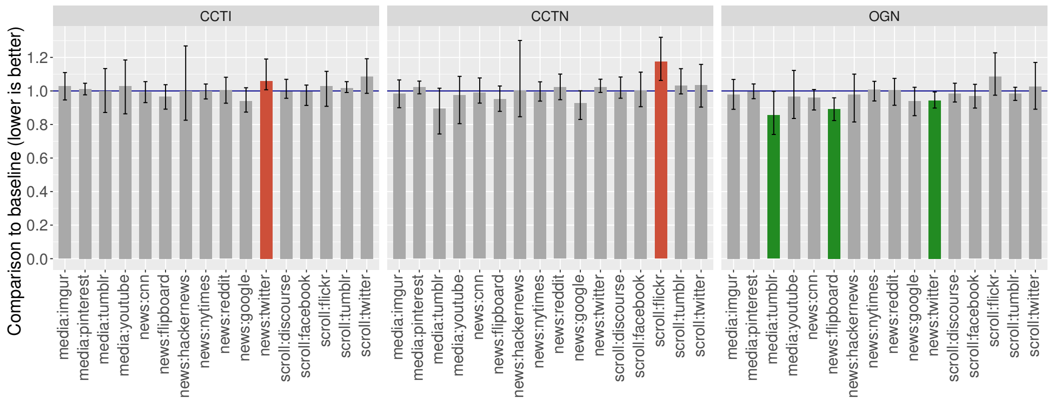


Fig. 14. Total garbage collection time compared to the OGI baseline

point 1.0 represents a result that is not statistically significant. Otherwise, the result is statistically significant and is either an improvement (the confidence interval is below 1.0) or a regression (confidence interval is above 1.0).

The CCTI configuration improves the pause times of 6 out of 16 webpages. Eight webpages do not show any statistically significant difference, and in two webpages the pause times increase. The improvement is the largest on *media:youtube*, where the average pause time decreases from 16ms to 10ms. On *scroll:tumblr* the average pause time increases from 10.3ms to 11.4ms, which is caused by Blink marking work discovered during finalization of garbage collection. The problem is that garbage collection transitions to finalization too early because the heuristics for starting finalization do not take into account the amount of remaining Blink marking work. We intend to improve the heuristics in future. As expected the CCTN and OGN configurations make the finalization pause times larger.

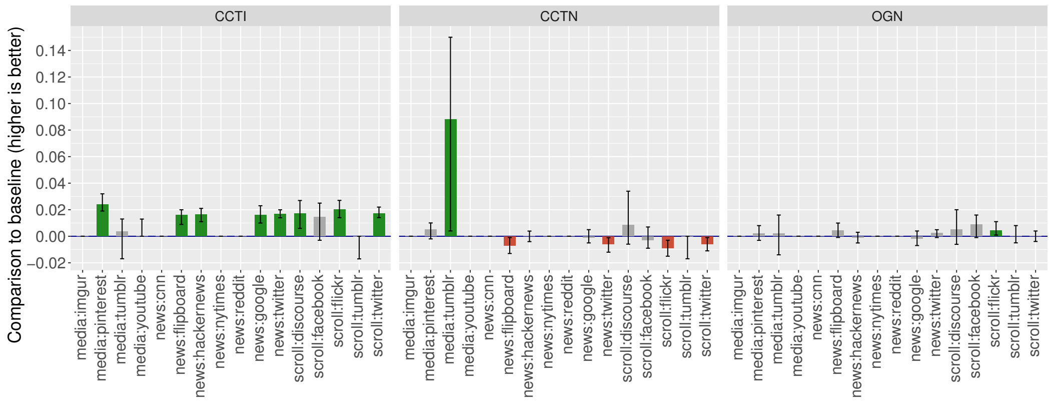


Fig. 15. Difference in MMU between a configuration and the OGI baseline with 16.67ms time window

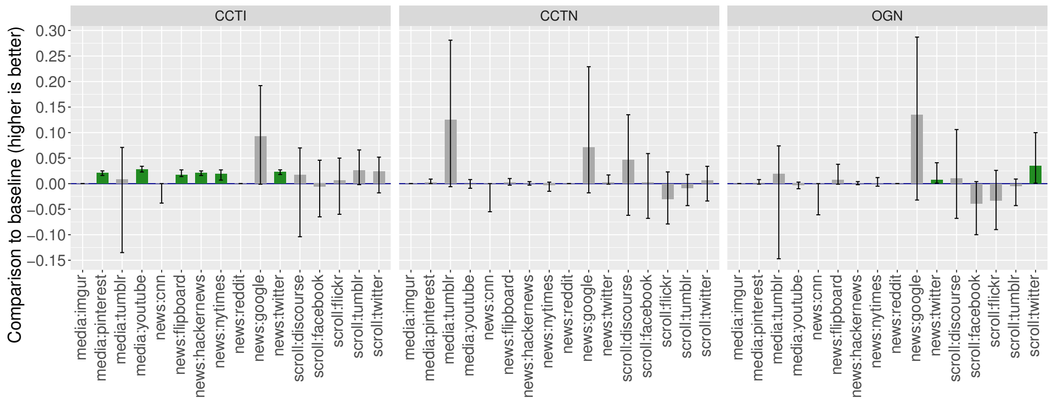
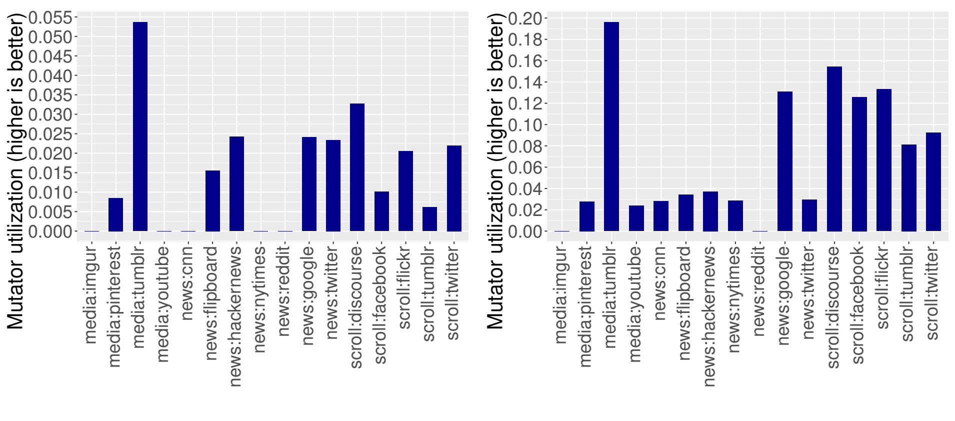


Fig. 16. Difference in MMU between a configuration and the OGI baseline with 50ms time window

Figure 14 compares the total garbage collection times between the configurations. The *news:twitter* page shows a statistically significant increase in total garbage collection time in the CCTI configuration: from 444ms to 480ms. This relatively small increase is expected and represents the trade-off of CCTI configuration.

The minimum mutator utilization is small for many webpages, as shown in Figure 17. Normalizing the MMU results relative to the OGI baseline might be misleading, sometimes suggesting huge improvements. We therefore report in Figure 15 and Figure 16 the difference between the MMU of a specific configuration and the OGI baseline MMU. A statistically significant result has the confidence interval that does not include 0.0. The CCTI configuration improves eight and six webpages with increases in MMU by 1% − 3% for the 16.67ms and 50ms time windows, respectively. The MMU change in CCTN and OGN configuration is mostly within the noise except for *media:tumblr* in CCTN configuration for the 16.67ms window. That workload is bimodal with MMU values around 3% in some runs and 13% in other runs. This suggests that there is non-determinism either in the webpage code or in Chrome that affects garbage collection scheduling.

Besides garbage collection times and MMU, we measured Chrome memory usage when running the benchmarks but we did not observe significant difference for all configurations. The memory



(a) 16.67ms time window (b) 50ms time window

Fig. 17. MMU of the OGI baseline

usage was within 5% of the OGI baseline. The underlying memory leaks described in bug reports[13]. are fixed though which should yield in better real-world memory usage.

7 CONCLUSION

In this paper we introduced the cross-component memory management problem that arises when embedding a virtual machine into another software system. Implementers face a choice: Design the system in a way where cycles are either avoided or are safely reclaimed. Avoiding cycles can be achieved by shifting the cross-component boundary until the object graph in one component is simple enough to provide a proof that cycles do not exist. In practice, this may be difficult and too restrictive, and implementers are left with providing general infrastructure to handle cross-component cycles.

We introduced such a general infrastructure with cross-component tracing (CCT), a garbage collection strategy that allows efficient, effective, and safe garbage collection over software component boundaries based on tracing. We showed that tracing over component boundaries is key to collect reference cycles which may otherwise result in memory leaks. We presented a generic algorithm for CCT that can be used as a guideline for other systems that require cross-component memory management and implemented CCT in Chrome's JavaScript engine V8 and its embedder the rendering engine Blink. When garbage collection is performed in V8, references between the JavaScript heap and the C++ Blink heap are traced in both directions to compute the precise transitive closure of all live objects spanning both components. To perform tracing efficiently, we integrated tracing through Blink's C++ heap into the incremental marking phase of the V8 garbage collector to keep pause times low which required incremental marking infrastructure like write barriers and step-wise visitation of objects in C++. We demonstrated in various real-world experiments that incremental CCT reduces pause times and improves mutator utilization in comparison to the previous (incremental) object grouping system of Chrome. CCT also eliminated memory leaks that are possible with object grouping. We also demonstrated how web developers can benefit from CCT by visualizing precise liveness and retainment information across V8 and Blink component boundaries in Chrome's heap snapshot memory tool. Incremental CCT replaced object grouping in

[13]https://crbug.com/501866

Chrome and has been enabled by default since version 57. CCT is also deployed in other widely used software systems such as Opera, Cobalt, and Electron.

We would like to conclude this paper with the observation that CCT comes with significant implementation overhead. Ultimately, implementers need to weigh off the effort of either avoiding cycles by enforcing restrictions or implementing a mechanism to reclaim cycles such as CCT.

ACKNOWLEDGMENTS

We would like to thank Ben L. Titzer for valuable feedback.

REFERENCES

M. Ager, E. Corry, V. Egorov, K. Hara, G. Wibling, and I. Zerny. 2013. Oilpan: Tracing Garbage Collection for Blink. (2013). Retrieved September 6, 2018 from https://docs.google.com/document/d/1y7_0ni0E_kxvrah-QtnreMlzCDKN3QP4BN1Aw7eSLfY

D. F. Bacon and V. T. Rajan. 2001. Concurrent Cycle Collection in Reference Counted Systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, London, UK, 207–235. https://doi.org/10.1007/3-540-45337-7_12

J. F. Bartlett. 1989. *Mostly-Copying Garbage Collection picks up Generations and C++*. Technical Note TN–12. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-12.pdf

H.-J. Boehm, E. Moss, J. Bartlett, and D. R. Chase. 1991. Panel Discussion: Conservative vs. Accurate Garbage Collection. Summary appears in Wilson and Hayes' OOPSLA'91 GC workshop report.

H.-J. Boehm and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

J. Brichau and C. De Roover. 2009. Language-shifting Objects from Java to Smalltalk: An Exploration Using JavaConnect. In *Proceedings of the International Workshop on Smalltalk Technologies (IWST '09)*. ACM, New York, NY, USA, 120–125. https://doi.org/10.1145/1735935.1735956

F. Brown, S. Narayan, R. S. Wahby, D. R. Engler, R. Jhala, and D. Stefan. 2017. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 559–578. https://doi.org/10.1109/SP.2017.68

C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 49–70. https://doi.org/10.1145/74878.74884

C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–678. https://doi.org/10.1145/362790.362798

P. Cheng and G. E. Blelloch. 2001. A Parallel, Real-time Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 125–136. https://doi.org/10.1145/378795.378823

Y. Chicha and S. M. Watt. 2006. A Localized Tracing Scheme Applied to Garbage Collection. In *Proceedings of the 4th Asian Conference on Programming Languages and Systems (APLAS'06)*, N. Kobayashi (Ed.). Springer-Verlag, Berlin, Heidelberg, 323–339. https://doi.org/10.1007/11924661_20

T. W. Christopher. 1984. Reference count garbage collection. *Software: Practice and Experience* 14, 6 (1984), 503–507. https://doi.org/10.1002/spe.4380140602

D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2887746.2754181

D. Clifford, H. Payer, M. Starzinger, and B. L. Titzer. 2014. Allocation Folding Based on Dominance. In *Proceedings of the 2014 International Symposium on Memory Management (ISMM '14)*. ACM, New York, NY, USA, 15–24. https://doi.org/10.1145/2602988.2602994

Cobalt Project Authors. 2018. Cobalt. Retrieved September 6, 2018 from https://cobalt.foo/

S. S. Craciunas, C.M. Kirsch, H. Payer, A. Sokolova, H. Stadler, and R. Staudinger. 2008. A Compacting Real-time Memory Management System. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 349–362.

U. Degenbaev, J. Eisinger, M. Ernst, R. McIlroy, and H. Payer. 2016. Idle time garbage collection scheduling. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 570–583. https://doi.org/10.1145/2908080.2908106

D. Detlefs. 1990. *Concurrent Garbage Collection for C++*. Technical Report CMU-CS-90-119. Pittsburgh, PA.

D. Detlefs. 1992. Garbage Collection and Run-time Typing as a C++ Library. In *In Proceedings of the 1992 Usenix C++ Conference*. USENIX Association, 37–56.

E. W. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. 1978. On-the-fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966–975. https://doi.org/10.1145/359642.359655

D. R. Edelson. 1992. A Mark-and-sweep Collector C++. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 51–58. https://doi.org/10.1145/143165.143178

S. Finne. 2016. Blink heap compaction. (2016). Retrieved September 6, 2018 from https://docs.google.com/document/d/1k-vivOinomDXnScw8Ew5zpsYCXiYqj76OCOYZSvHkaU

J. Geidel. 2018. JNIPort. Retrieved April 11, 2018 from https://sites.google.com/site/jniport/home

Google. 2018. Telemetry. Retrieved September 6, 2018 from https://github.com/catapult-project/catapult

M. Hablich and H. Payer. 2018. Lessons Learned from the Memory Roadshow. Retrieved September 6, 2018 from https://drive.google.com/file/d/1HyRDBgt6wvftC-_cne_3zuYbVtp1RoJD

R. H. Halstead. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)*. ACM, New York, NY, USA, 9–17. https://doi.org/10.1145/800055.802017

M. Hollander, D. A. Wolfe, and E. Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.

R. L. Hudson, R. Morrison, J. Eliot B. Moss, and D. S. Munro. 1997. Garbage Collecting the World: One Car at a Time. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, New York, NY, USA, 162–175. https://doi.org/10.1145/263698.264353

J. Hughes. 1985. A Distributed Garbage Collection Algorithm. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*. Springer-Verlag New York, Inc., New York, NY, USA, 256–272. http://dl.acm.org/citation.cfm?id=5280.5296

R. Jones, A. Hosking, and E. Moss. 2012. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall.

T. Kalibera and R. Jones. 2011. Handles Revisited: Optimising Performance and Memory Costs in a Real-time Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/1993478.1993492

R. Ladin and B. Liskov. 1992. Garbage collection of a distributed heap. In *Proceedings of the 12th International Conference on Distributed Computing Systems*. 708–715. https://doi.org/10.1109/ICDCS.1992.235116

Mozilla Contributors. 2011. MMgc. Retrieved September 6, 2018 from https://developer.mozilla.org/en-US/docs/Archive/MMgc

Mozilla Contributors. 2017. The XPCOM cycle collector. Retrieved September 6, 2018 from https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XPCOM/Interfacing_with_the_XPCOM_cycle_collector

NativeScript. 2018. Nativescript. Retrieved September 6, 2018 from https://www.nativescript.org

C. E. Oancea, A. Mycroft, and S. M. Watt. 2009. A New Approach to Parallelising Tracing Algorithms. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 10–19. https://doi.org/10.1145/1542431.1542434

Oracle Corporation. 2018. JNI Functions. Retrieved April 11, 2018 from http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html

F. Pizlo. 2017. Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector. Retrieved September 6, 2018 from https://webkit.org/blog/7122/introducing-riptide-webkits-retreating-wavefront-concurrent-garbage-collector/

D. Plainfossé and M. Shapiro. 1995. A Survey of Distributed Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management (IWMM '95)*. Springer-Verlag, London, UK, UK, 211–249. https://doi.org/10.1007/3-540-60368-9_26

Y. Shuf, M. Gupta, H. Franke, A. Appel, and J. P. Singh. 2002. Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/582419.582422

M. Slavchev. 2014. Synchronizing GC in Java and V8. Retrieved September 6, 2018 from https://iobservable.net/blog/2014/06/07/synchronizing-gc-in-java-and-v8

M. Slavchev. 2017. Memory management in NativeScript for Android. Retrieved September 6, 2018 from https://iobservable.net/blog/2017/11/04/memory-management-in-nativescript-for-android

S. Tilkov and S. Vinoski. 2010. Node.Js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (Nov. 2010), 80–83. https://doi.org/10.1109/MIC.2010.145

D. Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. ACM, New York, NY, USA, 157–167. https://doi.org/10.1145/800020.808261

V8 Project Authors. 2017. Fall cleaning: Optimizing V8 memory consumption. Retrieved September 6, 2018 from https://v8.dev/blog/optimizing-v8-memory

J. Vilk and E. D. Berger. 2018. BLeak: Automatically Debugging Memory Leaks in Web Applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3192366.3192376

D. R. White, J. Singer, J. M. Aitken, and R. E. Jones. 2013. Control Theory for Principled Heap Sizing. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2464157.2466481