Concurrent Marking of Shape-Changing Objects

Ulan Degenbaev
Google
Germany
ulan@google.com

Michael Lippautz
Google
Germany
mlippautz@google.com

Hannes Payer
Google
Germany
hpayer@google.com

Abstract

Efficient garbage collection is a key goal in engineering high-performance runtime systems. To reduce pause times, many collector designs traverse the object graph concurrently with the application, an optimization known as *concurrent marking*. Traditional concurrent marking imposes strict invariants on the object shapes: 1) static type layout of objects, 2) static object memory locations, 3) static object sizes. High performance virtual machines for dynamic languages, for example, the V8 JavaScript virtual machine used in the Google Chrome web browser, generally violate these constraints in pursuit of high throughput for a single thread. Taking V8 as an example, we show that some object shape changes are safe and can be handled by traditional concurrent marking algorithms. For unsafe shape changes, we introduce novel wait-free object snapshotting and lock-based concurrent marking algorithms and prove that they preserve key invariants. We implemented both algorithms in V8 and achieved performance improvements on various JavaScript benchmark suites and real-world web workloads. Concurrent marking of shape-changing objects using the wait-free object snapshotting algorithm is enabled by default in Chrome since version 64.

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

Keywords Garbage Collection, Memory Management, Runtime Environments, Language Implementation

ACM Reference Format:
Ulan Degenbaev, Michael Lippautz, and Hannes Payer. 2019. Concurrent Marking of Shape-Changing Objects. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3315573.3329978

1 Introduction

Many modern high-performance runtime systems employ garbage collection to reclaim unused application memory safely. A garbage collection cycle typically consists of three phases: 1) *marking*, where live objects are identified, 2) *sweeping*, where dead objects are released, and 3) *compaction*, where live objects are relocated to reduce memory fragmentation. Different collector designs may run these phases in sequence or combine them in various ways. This paper focuses solely on the marking phase.

During marking, the garbage collector (*marker* for brevity) 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. Pausing the application for the entire marking phase guarantees that the object graph is static. A static object graph simplifies traversal but may result in an unacceptably long pause. A standard technique to reduce marking pause is concurrent marking which allows the application (*mutator* for brevity) to run concurrently to multiple markers.

Concurrent marking is a well-established technique used in many popular runtime systems like Android Art [3], Go [24], and OpenJDK [15]. Traditional concurrent marking algorithms use three colors: *white*, *grey*, and *black*. White objects are not yet discovered and are potentially dead. Grey objects are discovered, but not yet visited. Black objects are visited and all their reference fields have been followed. A concurrent marking algorithm must ensure that when marking finishes all reachable objects are marked black and through that are kept alive. An invariant that ensures this correctness condition is the *strong tri-color invariant* [27] which requires that black objects do not reference white objects. During concurrent marking, the mutator may attempt to break the invariant by writing a white object reference into a field of a black object. This problem is solved with read or write barriers [27].

Traditional concurrent marking algorithms are designed with the assumption that the object graph is mutated only by field write operations. For high-performance dynamic language runtime systems, such a limitation is too restrictive and prohibits certain optimizations that rely on in-place changes of field types, object sizes, and object locations. We refer to such changes as *shape changes.* For example, consider a *double unboxing* optimization that replaces a reference to a number object with a raw floating-point number. The optimization results in a field type change if it is performed on an existing object. While such shape changes enable performance optimizations in a single-threaded environment, they require careful synchronization with concurrent garbage collection. For example, the marker may misinterpret a raw field as a reference field if it uses an old field type invalidated by concurrent mutator shape changes.

One example of a high-performance environment with shape changes is V8 – an industrial-strength open-source virtual machine for JavaScript [21]. V8 did not employ concurrent marking since its initial design more than 10 years ago. Instead, V8 has heavily exploited JavaScript's single-threaded execution model to pioneer a number of novel shape-changing runtime techniques. Such techniques benefit sequential performance but complicate concurrent marking.

In this paper we study the problem of concurrent marking in the presence of shape changes for environments with a single mutator thread. Taking V8 as an example, we show how shape changes appear in a virtual machine with sequential garbage collection as reasonable performance optimizations. We classify the shape changes as *safe* and *unsafe*. We show that safe shape changes can be performed without synchronization with the concurrent marker. For unsafe shape changes, we present and compare two concurrent marking algorithms. The first is a novel wait-free snapshotting algorithm. The second is a lock-based protocol that guarantees atomicity of unsafe shape changes. In addition to the tri-color invariant, we state and prove the reference safety invariant for the algorithms assuming C++-like memory model with acquire/release/relaxed atomic operations.

We evaluate the performance of our algorithms based on standard JavaScript benchmark suites and real-world workloads and show significant performance improvements and memory reductions in comparison to the V8 baseline collector configuration of sequential, incremental marking. We also show that the wait-free snapshotting algorithm results in less total marking CPU time in comparison to the lock-based versions since the markers and the main thread can make progress without waiting on each other.

The contributions of this paper are: 1) We present the problem of concurrent marking in the presence of shape changes. 2) We solve the problem with a generic wait-free snapshotting algorithm that relies only on synchronization on markbits and can be applied to other virtual machines. 3) We provide a concrete implementation based on the V8 JavaScript virtual machine and evaluate its performance.

2 V8 Overview

To make our presentation concrete we use the V8 JavaScript virtual machine as a running example of object shape changes and as an application of our general snapshotting algorithm. This section provides the background needed by introducing the relevant parts of V8 such as its object layout and garbage collector.

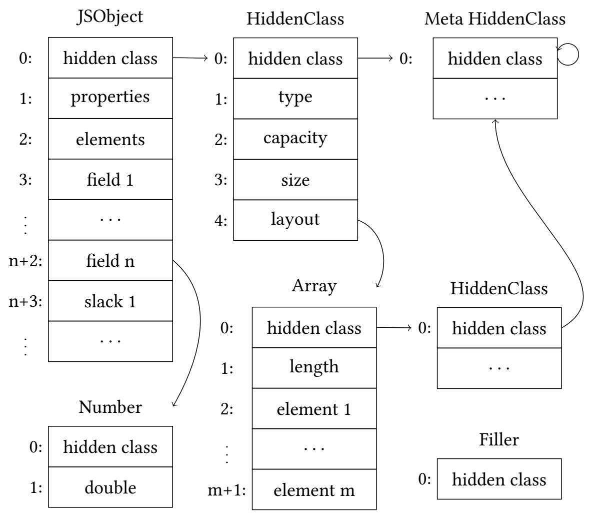


Figure 1. Simplified layout of arrays, JavaScript objects, hidden classes, and fillers.

2.1 Object Representation and Optimizations

Objects on V8's heap are allocated on word-aligned memory addresses and consist of multiple words. We refer to the words in an object as the object fields. The first field of an object points to a *hidden class* [2, 9] that describes the type, size, and memory layout of the given object. Hidden classes are allocated on the heap and look like ordinary managed objects. In particular, the first field of a hidden class points to the *meta hidden class*, which is its own hidden class. This object structure is depicted in Figure 1.

A hidden class describes whether an object field contains an object reference or a *raw value* of a different type like an integer, a floating point number, characters, etc. Since object references are word-aligned, the least significant bit of an object's address is guaranteed to be `0`. V8 repurposes the bit as a *tag bit* to unbox the *small integers* (SMI) into reference fields of an object. If the tag bit is `1`, then the value contains a pointer to an object. Otherwise, the value contains an SMI. In the rest of the paper we refer to values with the tag bit as *tagged values*. Thus, a tagged value is either a reference or an SMI. It is the hidden class that specifies whether a field stores a tagged value or a raw value. Note that without the context, a tagged value is indistinguishable from a raw value.

V8 has more than a hundred different object types. It is sufficient to focus on two fundamental types, `Array` and `JSObject`, each of which are subject to different kinds of shape changes. The `HiddenClass`, `Number` and `Filler` types are used in the paper but do not undergo shape changes.

2.1.1 Hidden Classes

We simplify our model of hidden classes to only the information necessary for garbage collection, ignoring JavaScript prototypes, constructors, etc. Our simplified hidden class consists of five fields as shown in Figure 1. The first field is a reference to the meta hidden class. The second field is a raw integer that encodes the instance type, i.e. the type of the objects that refer to this hidden class. The next two fields are raw integers that specify the capacity and the size of the instance objects. We define the capacity of an object as the total number of fields in the object and the size as the number of used fields. The unused fields at the end of the object are called *slack* fields. Only `JSObjects` have slack fields, so for all other types, the capacity equals the size. Note that `Arrays` have a separate field for storing their length, so their hidden classes have capacity and size `0`. The fifth field of a hidden class stores a reference to a bitmap which describes the object layout. A bit in the bitmap corresponds to a field in an instance object and indicates whether the corresponding field stores a tagged or raw value. The bitmap is only needed for `JSOject` instances since `Array`, `HiddenClass`, and `Filler` instances have static field types.

2.1.2 JavaScript Objects

`JSObjects` support dynamic addition and removal of user visible properties, a key feature of the JavaScript programming language. A property can be stored either in one of the fields of the object or in a separate backing store. As shown in Figure 1, a JavaScript object may have slack that is consumed when new properties are added. If slack is exhausted, then additional properties overflow to the backing store `Array`. Addition and removal of properties is accompanied with a change to a new `HiddenClass`. Adding or removing properties from an object is a size change. Properties are usually stored as tagged values. On 64-bit architectures, V8 uses a type change optimization, called *double-unboxing*, that allows numeric in-object properties to be stored as raw 64-bit floating-point numbers instead of boxed numbers. When this optimization is performed on an object, some of its fields that reference heap number objects transition from tagged to their raw double representation. The object header is overwritten to point to an appropriate new `HiddenClass` that reflects this transition in its layout bitmap.

2.1.3 Arrays

V8 has both tagged and raw `Arrays`, storing either all tagged elements or raw elements. The number of elements is stored in the second field as an SMI value.

`Arrays` are not directly exposed to the JavaScript application. They are used as backing store for `JSObjects` that implement JavaScript arrays (`JSArray`). A `JSArray` has a rich set of methods such as *push*, *pop*, *shift*, *unshift* that allow them to fill the roles of other basic data-structures like stacks and queues. For example, the push/unshift combination provides a queue, where the push method appends an element and the unshift method removes the first element.

Since `JSArrays` are ubiquitous in JavaScript applications, it is desirable to have the fastest possible implementation for the array indexing operation and the stack/queue operations mentioned above. The push/pop operations can be implemented in amortized $O(1)$ time using exponential backing store sizing. The unshift operation, however, is a little trickier because it deletes the `0`th element of an array and logically moves all elements down by `1`. V8 preserves fast array indexing by implementing this operation by shape changes in the form of in-place updates on the object header and length of the backing store array. V8 implements *array left-trimming* that allows unshift operations to move the start of the `Array` object in memory to its new location to ensure that the first element starts at index zero, which represents a location change. V8 also supports *array right-trimming* that shrinks an `Array` by decreasing its length, which is a size change.

2.1.4 Numbers and Fillers

A `Number` is a boxed floating point object. It has a static layout and consists of two fields: the hidden class reference and the floating-point number. `Filler` objects are even simpler, they only consist of a reference to their hidden class. Fillers are written into the space of shrunken objects to preserve the ability to iterate over the heap in order from low memory address to high memory address, which is used by certain non-critical VM tools.

2.2 Garbage Collection

V8 uses a generational garbage collection strategy with the heap separated into the young and the old generation. The young generation uses a semi-space strategy with bump-pointer allocation inlined in compiled code [10]. When the young generation's occupancy reaches a threshold, a parallel variant of Cheney's algorithm [7] collects the young generation. Objects which have been moved once in the young generation are promoted to the old generation.

The old generation also uses bump-pointer allocation on the fast path for tenured allocations [9]. Free old generation memory is organized in segregated free-lists, which are used to refill the current bump-pointer span. A full collection collects both the old and the young generation using a mark-sweep-compact strategy. V8 marks live objects incrementally in small steps to avoid long marking pauses. When marking finishes the *main pause* is executed consisting of several phases which concludes a garbage collection cycle: 1) Finalizing marking: This rescans the roots and continues marking until all reachable objects are marked black. 2) Starting sweeping: Sweeping is performed concurrently to the main thread. 3) Parallel evacuation of the young generation and old generation compaction for reducing memory fragmentation. After that the JavaScript execution is continued.

Orthogonal to the plain garbage collection optimizations V8 employs idle time garbage collection scheduling [13] and cross-component garbage collection to collect object graph cycles spanning the V8 and the C++ embedder heap boundary [14].

```
MemoryOrder = { acquire, release, relaxed }
⟨memory_order⟩? load ⟨variable⟩ ← ⟨address⟩
⟨memory_order⟩? store ⟨value⟩ → ⟨address⟩
⟨memory_order⟩? CAS (⟨expected⟩ → ⟨desired⟩) → ⟨
        address⟩
push ⟨address⟩ → ⟨worklist⟩
pop ⟨variable⟩ ← ⟨worklist⟩
```

Figure 2. Notation for memory and worklist operations.

2.2.1 Concurrent Marking

In this section we give a brief overview about the general concurrent marking infrastructure we added to V8. Our work takes advantage of the existing incremental marking infrastructure, i.e. object visitation mechanisms, marking bitmap, garbage collection scheduling heuristics, etc. Serial incremental marking on the mutator thread is used as the fallback to ensure marking progress in scenarios where no extra threads are available for markers. Incremental marking uses a Dijkstra-style write barrier [16] which we also use in concurrent marking to maintain the tri-color marking invariant.

Marking starts on the main thread at a safepoint, i.e. where the stack is in an iterable state, based on garbage collection scheduling heuristics [13]. At that point all the existing objects on the heap are white. The main thread scans the roots, marks discovered objects grey, and pushes them onto the marking worklist. Then it kicks-off multiple marking threads and resumes execution of JavaScript.

Each marker runs a loop that drains the marking worklist by popping an object from the worklist, marking it black, and visiting its reference fields. If a reference points to a white object, then the marker tries to color the object grey using an atomic CAS operation. If coloring grey is successful, then the object is pushed onto the marking worklist. Note that for clarity we are eliding details related to compaction which are not relevant for this paper (e.g. markers also collect field addresses that contain references to objects that will be subject to relocation in the compaction phase).

Markers may steal work from each other when they run out of work on their private worklists. The main thread periodically also helps with marking and checks if the main pause should be scheduled, i.e. when it is likely that all objects are marked.

In the following sections we describe how markers handle object shape changes.

3 Notation and Memory Model

We define the marker and mutator operations in pseudocode and want the pseudocode notation to be precise so that we can reason about safety and correctness of concurrent marking. In particular the notation should be precise about memory accesses, their ordering, and visibility. Since V8 is written

```
Types:
  Word = $\mathbb{B}^W$, where W = ⟨bits per word⟩
  AbsValueType = {Ref, Smi, Raw}
  Value = Word × AbsValueType
  ObjectType = {JS, HC, RA, TA, NU, FI}
  FieldType = {Tagged, Raw}
  Color = {white, grey, black}
Functions, globals:
  $\mathcal{A}$(_ × type) = type
  HasRefTag(word × _) = word & 1
  RefTag(ptr) = (ptr | 1) × Ref
  RefUntag(word × Ref) = word & ~1
  HasSmiTag(value) = not HasRefTag(value)
  SmiTag(smi) = (smi << 1) × Smi
  SmiUntag(word × Smi) = word >> 1
  Null = ⟨reference to the root null object⟩
  FillerClass = ⟨reference of the root hidden class of fillers⟩
```

Figure 3. Definitions of types, constants, and functions.

in C++, we will use the C++ memory model [5, 6]. We assume that the reader is familiar with the basic atomic operations and the *acquire*, *release*, and *relaxed* memory orders defined by the language standard.

Figure 2 shows the syntax of the operations that we will use in pseudocode. The memory operations map directly to the corresponding C++ *std::atomic* operations with one caveat: if the memory order is omitted, relaxed memory order is assumed. The compare-and-swap (CAS) operation corresponds to the *compare_exchange_strong* C++ operation.

The main challenge of our concurrent marking algorithms is to avoid misinterpreting reference fields. Since tagged and raw values are indistinguishable on the physical level without the context of the object shape, we need functionality that allows us to tell them apart on the abstract level, so that we can unambiguously state and prove the invariants of the algorithms. To do that we augment values with abstract field types as shown in Figure 3. The abstract type $\mathcal{A}(v)$ of a value v indicates whether the value is a reference, an SMI, or a raw value. We assume that the abstract type is preserved by memory operations that load and store the value. Since $\mathcal{A}$ is merely a tool for proofs and invariants, neither the mutator nor marker can use it.

The *Tag/Untag* functions in Figure 3 implement the SMI tagging scheme described in Section 2.1. We use concise syntax for computing the addresses of object fields. Let *obj* be a reference to an object, which means that $\mathcal{A}(obj) = Ref$. Then $obj[i]$ denotes the address of the i-th field of the object. More precisely, $obj[i] = RefUntag(obj) + i$, assuming that the address arithmetic is carried out on the word granularity.

Without diving into implementation details we assume two data-structures as given: the marking worklist and the markbits. The marking worklist is a thread-safe collection

```
24 marker DrainWorklist():
25   while (obj ← pop Worklist):
26     ⟨synchronize with the initializing stores of obj⟩
27     VisitObject(obj)
28
29 marker VisitObject(obj):
30  acquire load class ← obj[0]
31  load type ← class[1]
32  switch (type):
33    case JS: VisitJSObject(obj, class)
34    case HC: VisitHiddenClass(obj, class)
35    case RA: VisitRawArray(obj, class)
36    case TA: VisitTaggedArray(obj, class)
37    case NU: VisitNumber(obj, class)
38    case FI: impossible
```

Figure 4. Marking loop and visitor dispatch.

of object references that can be added and removed with the *push* and *pop* operations. Each object has two mark-bits associated with it that encode three color values: white, grey, and black. For accessing mark-bits, we assume that there is a function *Markbits*(obj) that given a reference to an object returns the address of its mark-bits. This allows us to use the same syntax for accessing object fields and markbits. Depending on the actual implementation, the mark-bit accessors might be replaced with more complex operations, e.g. with operations that mask out other marking bits if multiple bits are packed into a single byte.

For defining the mutator and marker operations we will use the common statements and operators such as *if-then-else*, *for-in*, *let* that should be self-explanatory. Note that the first word before the operation name indicates whether the operation is performed by the mutator or the marker.

4 Basic Marker and Mutator Operations

This section sets up the scene by describing the basic operations of the marker and mutator performed during the marking phase of garbage collection. The operations do not include shape changes, which will be covered in Section 5. We show how the marker processes the marking worklist and follows discovered references. On the mutator side, we spell out our assumptions about object allocation and give a precise definition of the write barrier. Along the way, we state invariants about worklist items, object and hidden class fields, and reference safety.

4.1 Draining Marking Worklist

The marker runs the worklist draining loop depicted in Figure 4. In each iteration, the marker gets an object from the worklist, loads its hidden class, looks up the object type from the hidden class, and dispatches to the specialized visit function based on the object type. Let's refer to that object as the *currently visited object* (CVO). The following invariant ensures that the reference to the CVO obtained from the worklist is valid.

Invariant 4.1 (Worklist Items)**.** *Each worklist item is a reference to an object. Formally,* $\forall v \in Worklist.\mathcal{A}(v) = Ref$*.*

The invariant is maintained by the mutator when it pushes references onto the worklist during the initial root scanning and during the invocation of the write barrier. The marker pushes onto the worklist when it follows a discovered reference. As we will see later on, Invariant 4.4 guarantees that only references are pushed onto the worklist.

The marker may load any field of the CVO. We need to ensure that the marker never sees uninitialized fields.

Invariant 4.2 (Initialized Fields)**.** *Loading any field of the CVO returns a value which was written by the mutator at or after the initialization of the CVO.*

Line 26 in Figure 4 is the place where the marker synchronizes with the mutator to establish Invariant 4.2 while relying on Invariant 4.1. Efficient synchronization with the initializing stores of an allocated object is an interesting problem on its own and is out of scope of this paper as it would pull in all the details about object allocation and initialization. Thus we assume that there is some synchronization and take Invariant 4.2 as granted.

The hidden class reference load in Line 30 is now justified by Invariants 4.1, 4.2. However, loading the type field from the hidden class can go wrong if the hidden class was recently allocated and installed on the object by a shape change operation. In that case, the loaded type might be an uninitialized value. We need an invariant that ensures that all fields of the hidden class of the CVO are initialized.

Invariant 4.3 (Initialized HiddenClass Fields)**.** *Loading any field of the hidden class of CVO returns a value which was written by the mutator at or after the initialization of the hidden class.*

As we will see in Section 5, all object shape changing operations write to the hidden class field of an object using a release-store operation. Since Line 30 loads the hidden class using an acquire-load operation, the invariant follows from the memory model.

We assume that the specialized visitation functions like *VisitJSObject* depicted in Figure 4 know how to find all tagged fields of the object based on its hidden class and size if it is guaranteed that the object is not undergoing a concurrent shape change. Later on in Sections 6 and 7 we will modify the *VisitObject* function to lift the shape change restriction.

Young generation garbage collection pauses concurrent marking threads and updates all references in the marking worklist after object relocation.

```
40 marker FollowReference(value):
41   if HasRefTag(value):
42     load color ⟵ Markbits(value)
43     if color = white and CAS (white ⟶ gray) ⟶ Markbits(
             value)
44       push value ⟶ Worklist
```

Figure 5. Shade an object to grey and push onto the worklist.

```
45 mutator WriteField(obj, i, value, type):
46   store value ⟶ obj[i]
47   WriteBarrier(value, type)
48
49 mutator WriteBarrier(value, type):
50   if type = Tagged and HasRefTag(value):
51     load color ⟵ Markbits(value)
52     if color = white and CAS (white ⟶ grey) ⟶ Markbits(
             value):
53       push value ⟶ Worklist
```

Figure 6. Write field with Dijkstra-style write barrier.

4.2 Following References

With the help of the specialized visit function, the marker iterates the tagged fields of the object. For each tagged field, the marker loads its value and checks whether the value is a reference or an SMI, as shown in Figure 5. If the value is a reference to a white object, then the marker tries to shade the object grey with an atomic CAS operation. If that succeeds, then the reference is pushed onto the marking worklist.

Note that the *HasRefTag* check in Line 41 assumes that the value is a tagged value. If the mutator changes the field type concurrently and confuses the marker to load a raw value instead, then the *HasRefTag* check may succeed and the marker may try to access mark-bits using an invalid reference. This brings us to the central invariant in the paper.

Invariant 4.4 (Reference Safety)**.** *The value passed to the FollowReference is a tagged value, that is* $\mathcal{A}(value) \in \{Ref, Smi\}$.

In Sections 6 and 7 we will show that this invariant holds for two algorithms in the presence of object shape changes.

4.3 Write Barrier

Having described the basic marker operations and invariants, we switch to the mutator. The most common object graph mutating operation is a write to an object field. Figure 6 defines the operation. We assume that this operation is only executed for the fields of an object that do not control the object shape and that it preserves the given field type.

We use a Dijkstra-style write barrier to maintain the tricolor marking invariant [27]. Note that to avoid synchronization with the marker the write barrier does not check the color of the source object. Thus, a write of a white object reference into a field of a white object marks the referenced object grey. This may result in floating garbage, but is crucial for performance as it avoids a memory fence after each write.

Shape Change	s'	u'	e'
Field Type Change	-	-	-
Append Field	-	$u + 1$	-
Remove Field	-	$u - 1$	-
Right Trim	-	$min(u, e - 1)$	$e - 1$
Left Trim	$s + 1$	-	-

Figure 7. Effect of a shape change on the object start, the slack area start, and the object end.

Shape Change	f'
Field Type Change	$f[k \mapsto v]$ for some (k, v): $s + h(t) \leq k < u$, $v \in FieldType$
Append Field	$f[u \mapsto v]$ for some v: $v \in FieldType$
Remove Field	$f[(u - 1) \mapsto \epsilon]$
Right Trim	$f[(e - 1) \mapsto \epsilon]$
Left Trim	$f[s \mapsto \epsilon, i \mapsto f(i - 1)]$, $\forall i : s < i \leq s + h(t)$

Figure 8. Effect of a shape change on field types.

5 Object Shape Changes

Generalizing V8's object representation, we model an object shape as a tuple (t, f, s, u, e), where t is the type of the object, f is a mapping from addresses to $FieldType \cup \{\epsilon\}$, s is the start address of the object, u is the start address of the slack area in the object, e is the end address of the object. All addresses are word-aligned. For brevity, let's assume that address arithmetic is carried out on the word granularity. Thus, for example, $s + 1$ is the address of the second field. Then the capacity of the object is equal to $e - s$ and the object size is $u - s$.

An object shape must satisfy certain constraints to be valid. For example, the size cannot not exceed the capacity. Also, the size cannot be smaller than the minimum object size, which we denote as $h(t)$ for the given object type t. Formally, the two constraints translate to $s + h(t) \leq u \leq e$. The field type function indicates whether a used field of the object is tagged or raw. It is set to ϵ for all slack fields and all fields outside the object. Thus, $\forall s \leq i < u : f(i) \in FieldType$ and $\forall i < s \vee i \geq u : f(i) = \epsilon$.

Having defined the object shape, we can now enumerate primitive shape changes by considering changes in the individual components of the object shape that result in valid object shapes: $(t, f, s, u, e) \rightsquigarrow (t', f', s', u', e')$. The shape changes that our algorithms can handle are listed in Figures 7 and 8 along with their effects on (s, u, e) and f. For the latter, we use the function update syntax $f[k \mapsto v]$, which means that the new function is the same as f except for the argument k that maps to v. The listed shape changes can be combined and repeated arbitrarily to produce more complex shape changes.

Note that the list is missing three other possible primitive shape changes: object type change ($t' \neq t$) and two object expansion changes ($s' < s$ and $e < e'$). We are not aware of any practical applications of an object type change. Object expansion is performed in a limited way by the Spidermonkey JavaScript VM[29]. When expanding an object the mutator needs to be careful to not expand into the next or the previous object; it may require in-place updates of free-list entries, and invalidate marking worklist entries, which comes with performance and memory overhead.

The mutator performs a shape change operation on an object by writing to its fields. We classify the writes into four kinds: 1) writes to the shape-controlling fields of the object (the hidden class reference field or the length field), 2) writes to the fields that changed their type. 3) clearing writes to the removed fields that became slack fields. 4) clearing writes to the trimmed fields that are no longer part of the object.

In order to support the marker, the mutator places a write barrier after each write of a reference. Additionally, the Hidden Class Invariant 4.3 requires that the write to the hidden class field is performed with a release-store operation. The value written by the clearing writes (kind 3, 4) must be tagged. Specifically, a removed field is cleared with the *Null*, which is a reference to a special pre-allocated root object, and a trimmed field is cleared with the *FillerClass*, which is a reference to the root hidden class of fillers. Note that trimming a field effectively allocates a filler object. This is important for the left trim operation, which moves the object start. If it would not leave a filler object at the old start, then the existing reference to the left-trimmed object would become invalid, breaking the Worklist Invariant 4.1. Since the mark-bits are associated with the object start, the left trim operation also copies them to the new object start.

Figure 9 shows a few examples of possible shape change operations that are derived by repeating simple shape changes and are specialized for JavaScript objects and arrays. Many other shape change operations can be obtained by combining the simple shape changes. Our marking algorithms support them in generic way.

5.1 Safe and Unsafe Shape Changes

Some shape changes work with concurrent marking without any special synchronization. We call such shape changes safe.

```
mutator FieldTypeChange(obj, new_class, affected_fields):
  release store new_class → obj[0]
  WriteBarrier(new_class, Tagged)
  for (index, value, type) in affected_fields:
    store value → obj[index]
    WriteBarrier(value, type)

mutator LeftTrimArray(obj, old_length, new_length):
  load color ← Markbits(obj)
  k = old_length − new_length
  store color → Markbits(obj + k)
  load class ← obj[0]
  release store class → obj[k]
  WriteBarrier(class, Tagged)
  store new_length → obj[k + 1]
  for i in [0, k):
    store FillerClass → obj[i]
```

Figure 9. Examples of shape change operations.

To understand what makes a shape change safe, consider a sequence of shapes and shape changes of one object since the start of marking:

$$shape_0 \xrightarrow{change_1} shape_1 \xrightarrow{change_2} \dots \xrightarrow{change_n} shape_n$$

A sequence of shape changes is safe if it does not matter in which order the marker sees the stores emitted by the shape changes and by the ordinary *WriteField* operations between the shape changes. For example, a sequence of right-trim operations is safe because even if the marker observes the cleared field value before the size change and follows the cleared value as a reference, it will arrive at a valid object (*FillerClass*). This is generalized and formalized in the following sufficient condition for safe shape changes.

Definition 5.1 (Safe shape changes)**.** Let $(t_i, f_i, s_i, u_i, e_i)$ denote the i-th shape in a sequence of shape changes $0 \leq i \leq n$. We call the sequence safe if:

1. each change updates a single shape-controlling field (either the hidden class reference or the length field).
2. all changes update the same shape-controlling field.
3. $\forall 0 \leq i, j \leq n : s_i = s_j$
4. $\forall a \in \mathbb{B}^W, a \bmod W = 0 : \{Tagged, Raw\} \not\subset types(a)$, where $types(a) = \bigcup_{i=0}^{n} f_i(a)$

The first three conditions are about atomicity of shape-controlling field updates and ensure that the marker is guaranteed to see coherent shapes without synchronization. Note that the third condition states that the object start does not move. The fourth condition requires that an object field does not appear as a tagged field in one shape and as a raw field in another shape. Note that ϵ is allowed in *types*(a), so a field can be a tagged field in one shape and can become a slack or trimmed field in another shape.

Theorem 5.2 (Reference safety with safe shape changes)**.** *If an object undergoes only safe shape changes, then the Reference Safety Invariant 4.4 holds for the marker visiting the object.*

Proof. Let f_i denote the field type function of the i-th shape in the sequence of shape changes ($0 \leq i \leq n$). Since all shape changes update the same shape-controlling field, the marker sees the stores to this field in the issued order. Thus we can assume that the marker loaded the field value at the j-th shape and iterates object fields according to f_j. (Note that this assumes that if the loaded value is a reference, then the referenced structure, e.g. hidden class, is immutable).

Consider some field of the object at address a. Let v be the value loaded by the marker from this field. Let k be the index of the shape in the sequence that the object had when v was written (Invariant 4.2 guarantees that the value comes from one of the mutator writes). Note that the mutator ensures that $\mathcal{A}(v)$ is consistent with $f_k(a)$.

Now we have several cases based on $f_j(a)$ and $f_k(a)$.

1. $f_j(a) = Raw \lor f_j(a) = \epsilon$: impossible because the marker iterates only tagged fields according to f_j.
2. $f_j(a) = f_k(a) = Tagged$: the value v is also tagged because the mutator keeps field types and value types consistent.
3. $f_j(a) = Tagged \land f_k(a) = Raw$: impossible due to the fourth condition in Definition 5.1.
4. $f_j(a) = Tagged \land f_k(a) = \epsilon$: the value v is the value of a slack field (*Null*) or a trimmed field (*FillerClass*). Both of them are tagged.

Thus if the marker follows a field value, then it is tagged. □

Now we identify a subset of simple shape changes that is guaranteed to produce safe shape change sequences. We can rule out the field type change and the left-trim shape changes as unsafe right away.

The remaining shape changes, the append field (AF), the remove field (RF), and the right-trim (RT), are safe for JavaScript objects in isolation. For example, any sequence of AF operations is safe. However, combining different shape changes is not necessarily safe. For example, an append of a tagged field, followed by a remove of the field and an append of a raw field results in a field that appears as both tagged and raw in the shape sequence. Thus the combination of AF/RF is unsafe. The remaining two combinations, AF/RT and RF/RT, are safe in isolation.

We conclude the section by proving the tri-color invariant.

Theorem 5.3 (Tri-color invariant with safe shape changes)**.** *If an object undergoes only safe shape changes, the tri-color invariant holds for the object.*

Proof. Each shape change operation emits a write barrier for each written references. Thus, if a field transitions from ϵ to *Tagged*, the write barrier maintains the tri-color invariant for it. If a field transitions from *Tagged* to ϵ, then the tri-color invariant holds for it trivially since the reference is removed from object. If a field stays *Tagged* for all shape changes, then the marker follows it no matter which shape it uses to iterate the tagged fields. □

```
71 mutator UnsafeShapeChange_S(obj, ...):
72   load color ← Markbits(obj)
73   if color ≠ black:
74     CAS (white → grey) → Markbits(obj)
75     if acquire CAS (grey → black) → Markbits(obj):
76       VisitObject(obj, class)
77   UnsafeShapeChange(obj, ...)
```

Figure 10. The mutator side of the snapshotting algorithm.

6 Snapshotting Algorithm

The main idea of our snapshotting algorithm is to synchronize the mutator and the concurrent marker using the markbits of the object that is undergoing a shape change. Before performing a shape change operation, the mutator ensures that the object is black, which means that if the object is white or grey, then the mutator tries to mark it black with an atomic *acquire-CAS* operation and on success visits it as shown in Figure 10.

On the other side, the concurrent marker visits the object as shown in Figure 11. (The *VisitObject_S* is the snapshot version of the *VisitObject* and replaces it in the worklist draining loop of the marker.) First the marker tries to take a snapshot of all tagged fields of the object. For this it inspects the hidden class of the object and loads the length field if the object is an array. If the hidden class is a *FillerClass* reference or the length is not a tagged integer, then the object was left-trimmed and can be safely skipped because the mutator ensures that the object is visited before left-trimming it. Otherwise, the marker knows the size of the object and loads all the tagged fields of the object into a local snapshot buffer.

After that, the marker validates the snapshot by marking the object black with an atomic *release-CAS* operation. If the operation fails, the object can be safely skipped because it is visited by the mutator or another marking thread. Otherwise, the marker follows the references in the snapshot.

Theorem 6.1. *The snapshotting algorithm maintains the Reference Safety Invariant 4.4.*

Proof. The *FollowReference* in Line 82 is invoked only if the *release-CAS* in Line 80 succeeds. The *release-CAS* synchronizes with the *acquire-CAS* in Line 75 on the mutator side and ensures that the snapshot loads happen before any store of an unsafe shape change operation. This means that the snapshot is taken on the object that had only safe shape changes. Theorem 5.2 ensures that the values in the snapshot are tagged. □

```
78  marker VisitObject_S(obj):
79    let (snapshot, success) = TakeSnapshot(obj)
80    if success and release CAS (grey → black) → Markbits(
          obj):
81      for value in snapshot:
82        FollowReference(value)
83
84  marker TakeSnapshot(obj):
85    acquire load class ← obj[0]
86    if class = FillerClass:
87      return ([], false)
88    load type ← class[1]
89    let (size, success) = FetchSize(obj, class, type)
90    if not success:
91      return ([], false)
92    let tagged_fields = TaggedFields(obj, class, type, size)
93    let snapshot = []
94    for i in tagged_fields:
95      load value ← obj[i]
96      snapshot.append(value)
97    return (snapshot, true)
98
99  marker FetchSize(obj, class, type):
100   if type = TA or type = RA:
101     load size_tagged ← obj[1]
102     if not IsSmiTag(size_tagged):
103       return ([], false)
104     return SmiUntag(size_tagged)
105   else:
106     load size ← class[3]
107     return size
108
109 marker TaggedFields(obj, class, type, size):
110   ⟨Returns a list of tagged field indices of the object.⟩
```

Figure 11. The marker side of the snapshotting algorithm.

The tri-color invariant follows similarly from Theorem 5.3 since the mutator visits the object if it marked it black before an unsafe shape change.

The algorithm is wait-free [22] because a marking thread can make progress independent from the mutator (and other marking threads) and vice-versa. We assume here that the CAS operations on markbits are wait-free, which is the case in V8, i.e. a CAS can only fail a bounded number of times due to forward marking progress. The simplicity and wait-freedom of the algorithm comes at the cost of requiring the mutator to visit a non-black object before performing an unsafe shape change operation on it. Note that once the object is visited subsequent unsafe operations do not have that overhead. Thus the total overhead of the algorithm depends on the number of different objects undergoing unsafe operations, not the total number of unsafe operations.

```
111 marker VisitTaggedArray_S(obj, class):
112   load length ← obj[1]
113   if release CAS (gray → black) → Markbits(obj):
114     FollowReference(class)
115     for i in [0, SmiUntag(length)):
116       load value ← obj[2 + i]
117       FollowReference(value)
```

Figure 12. Optimized snapshotting of arrays.

6.1 Optimized Array Snapshot

Since arrays can be arbitrarily large, it is undesirable from the performance and the memory usage point of view to snapshot all array elements. If we disallow transitions between tagged and raw arrays, which is the case in V8, then we can get O(1) snapshot for arrays. Recall that tagged arrays have only tagged elements, and raw arrays have only untagged elements. This means that visiting a raw array is as simple as following its hidden class reference, which does not require snapshotting.

Since tagged arrays cannot transition to raw arrays, we only need to snapshot the length of the array as shown in Figure 12. (Assume that the function is called by the *VisitObject_S* if the type field of the hidden class indicates an array.) This works because array trimming keeps all the fields tagged as it overwrites the fields with either a tagged *FillerClass* references, or a tagged array hidden class reference, or a tagged array length. So the value loaded in Line 116 is guaranteed to be tagged.

7 Lock-based Algorithm

As a baseline for performance comparison we implement the simplest possible lock-based algorithm that prevents an unsafe shape change of an object while the marker is visiting it. The idea is to require the marker and the mutator to lock the object before performing an operation on it. We abstract the implementation and the granularity of the lock with the *lockimp(obj)* function that returns a lock based on the object address. Section 9 evaluates performance of the algorithm with fine-grained per-word locks and coarse-grained per-page locks. A V8 page is a chunk of memory of 512K.

Locking on the mutator side is simple as Figure 13 shows. The marker, however, has to account for the left-trim operation moving the object start while it is waiting for the lock. Thus, after locking the marker checks whether the object is a filler. If so, then the object was left-trimmed in the meantime. In that case, the marker marks the filler black, releases the current lock, and re-tries locking at the next word as shown in Figure 14. Otherwise, the marker can safely visit the object and is guaranteed that an unsafe change will not happen concurrently. Proofs of the reference safety and tri-color invariants are left out due to space limitations.

```
mutator UnsafeShapeChange_L(obj, ...):
  lock(lockimpl(obj))
    UnsafeShapeChange(obj, ...)
  unlock(lockimpl(obj))
```

Figure 13. The mutator side of the lock-based algorithm.

In the lock-based algorithm the mutator thread cannot make progress if a marker thread locks an object which is needed by the mutator. In Chrome responsiveness of the mutator thread is critical e.g. to handle high priority input events or compute animations. Hence, we do not consider the lock-based algorithm a viable choice for Chrome.

Let us now proof correctness of the reference safety and tri-color invariants.

Theorem 7.1. *The lock-based algorithm maintains the Reference Safety Invariant 4.4.*

Proof. When the marker obtains the lock on obj, it is guaranteed by the lock that the mutator is not concurrently performing an unsafe operation that changes $obj[0]$.

The mutator ensures that no object refers to a filler, so fillers in the worklist can appear only after left-trimming. So if $obj[0] = FillerClass$ then the mutator must have left-trimmed the object before. In that case, the marker releases the current lock, advances by the size of the filler and locks the new address. Since the mutator can left-trim an object finite number of times, the marker at some point obtains a lock and sees a non-filler object.

The lock guarantees that the mutator is not performing any unsafe shape change on that object concurrently (including left-trimming). Since releasing the lock on the mutator side synchronizes with acquiring it on the marker side, the marker observes all stores until the last unsafe shape change performed on the object.

There could have been safe shape changes since the last unsafe shape change. Now we consider the sequence of these safe shape changes and repeat the steps in the proof of Theorem 5.2 to get the Reference Safety invariant. □

The tri-color invariant follows similarly by repeating the steps of the proof of Theorem 5.3.

8 Related Work

Concurrent marking is implemented in many production and research virtual machines. The OpenJDK G1 garbage collector [15] is based on the snapshot-at-the beginning algorithm [35] with black allocation for newly allocated objects which ensures the tri-color marking invariant. The Go garbage collector [24] is inspired by Sapphire [26]. It uses a hybrid write barrier [25] that combines a Yuasa-style [35] and Dijkstra-style [16] write barrier which guarantees together with black allocation marking progress. Android Art [3] uses

```
marker VisitObject_L(obj):
  do:
    lock(lockimpl(obj))
      load class ← obj[0]
      if class = FillerClass:
        store black → Markbits(obj)
        let obj = PtrTag(obj[1])
      else:
        VisitObject(obj)
    unlock(lockimpl(obj))
  while class = FillerClass
```

Figure 14. The marker side of the lock-based algorithm.

a Baker-style read barrier [4] which ensures that all objects whose references are read are kept alive. The C4 garbage collector [34] is using a similar read barrier for marking. Unsafe shape changes are avoided in these systems.

Object shape changes are not common in statically typed languages. In dynamically typed languages, where shape changes are to some degree part of the programming language, like in JavaScript, it may be necessary to allow them as part of the virtual machine design. There is not much literature about other JavaScript virtual machine implementations. The JSC JavaScript virtual machine implements concurrent marking and field type changes are discussed in a blog post [32]. The authors use an obstruction-free snapshot protocol proposed earlier [1] that may be subject to infinite retries and lock-based solutions for other complicated shape changes. A general overview of generic snapshotting mechanisms can be found in [33]. ChakraCore [36] implements concurrent marking but details about shape changes and the actual implementation are not documented. Spidermonkey [17] does not employ concurrent garbage collection but uses various shape changes like left-trimming [30] to optimize sequential performance.

A lock-based approach to guard shape changes (also called layout changes) was proposed in [11]. This work was later improved in [12] that introduced a *lightweight layout lock*, which allows to concurrently read an object without taking a lock in the fast path and falls back to locking in the slow path. It can be viewed as a hybrid of our snapshotting algorithm and the lock-based algorithm.

9 Evaluation

We compare five different marking configurations: 1) Incremental marking (I) is the baseline since it was the default configuration in Chrome until version 63. 2) Plain concurrent marking without snapshotting with unsafe shape changing optimizations being disabled (N). We use this configuration to evaluate the overhead of handling unsafe shape changes. Moreover, we will see the performance impact of the disabled optimizations. 3) The wait-free snapshot-based concurrent marking algorithm (*S*) which is the default since Chrome 64. 4) Lock-based concurrent marking using per-word locks (*LW*). 5) Lock-based concurrent-marking using per-page locks (*LP*). Our main focus is on configuration *S* and we use the other configurations to explore interesting garbage collection tradeoffs.

9.1 Benchmarking Setup

We use a development version of Chrome 72 (revision: `43dcdf8ccdde`) with patches for the marking variants for *N*, *LW*, and *LP*. The incremental marking configuration *I* can be configured with standard V8 start-up flags. We fix V8's young generation to 4MiB, the heap growing factor to 30%, and we disable garbage collections in idle time [13] to reduce sources of non-deterministic garbage collection scheduling. All benchmarks are run on a Linux workstation with two Intel Xeon E5-2690 V3 12-core 2.60 GHz CPUs and 64GB of main memory.

We evaluate each marking configuration on JavaScript and real-world macro-benchmarks using Chrome's tracing infrastructure and the Catapult performance testing framework [18]. Catapult runs scripted website interactions in Chrome and records all server responses, enabling predictable replay.

For macro-benchmarks we use the official real-world website benchmark sets of Chrome that mimics common interactions like scrolling, social network browsing, news website browsing, and media website browsing. The used workloads with URLs and their workload definition can be found in the Chrome's repository [19]. We run the following ten workloads: *tumblr*, *cnn*, *flipboard*, *reddit*, *google-{1,2}*, *nytimes*, *twitter-{1,2}*, and *discourse*.

For JavaScript benchmarks we use locally patched versions of *Octane 2.0* [20] and *Speedometer 2.0* [31] to provide tracing information which allows us to reason about garbage collection pause times and memory consumption.

9.2 Metrics

We are interested in benchmarking garbage collection effectiveness, efficiency, latency, and managed heap memory usage. For effectiveness and efficiency we measure cumulative marking time on the main thread and all threads, respectively. For latency we provide a minimum mutator utilization (MMU) metric [8]. MMU is the fraction of time of a given time window size where the main thread was paused the longest. We measured and compute window sizes from 1ms to 1000ms. We present window sizes of 16.67ms and 50ms as they represent the duration of one frame in a 60 frame-per-second animation and the suggested response time in Chrome's RAIL model [28], respectively. A MMU equal to 1 is the best possible result and an MMU equal to 0 is the worst possible result. Memory is measured by collecting periodic samples of the dirty memory size of the renderer process.

We run each benchmark for each marking configuration ten times, retrieving ten samples per metric. *I* is the baseline and we compare the other configurations with it using Wilcoxon rank sum test [23]. We report results with the 95% confidence intervals.

A result is not statistically significant (colored in grey) if the confidence intervals include the data point 1.0 (0.0 for non-normalized comparisons). Confidence intervals strictly above or below 1.0 (0.0 for non-normalized comparisons) represent statistically significant regressions (colored in dark grey) or improvements (colored in white), respectively.

9.3 Results

Figure 15 compares normalized cumulative marking time on the main thread with *I*. All concurrent configurations (*N*, *S*, *LW*, *LP*) improve over the existing baseline on all measured benchmarks, ranging from around 35% on `reddit` to 100% improvement on `tumblr`. The differences between concurrent configurations are not significant.

Figure 16 compares normalized cumulative marking time on all threads (main and background) with *I*. All concurrent configurations spend more time on marking objects, which is expected as that shows the synchronization overhead of the concurrent algorithms. The main result is that *S* is more efficient than *LW* which is more efficient than *LP*. The provided wait-freedom of *S* seems to pay off. The configuration *S* is on par with *N* showing that the overhead of the snapshot-based marking algorithm is negligible.

Note that Figure 15 and 16 shows that optimizing for performance on the main thread comes with a synchronization cost that increases cumulative marking time on all threads. For virtual machines with a single-threaded mutator like V8, reducing garbage collection time on the main thread has the highest priority as it directly affects user experience.

Figure 17 compares normalized renderer memory consumption with *I*. The measurements include managed memory (JavaScript/DOM objects) as well as unmanaged memory (bookkeeping memory, markbits, lockbits, etc). We want to evaluate whether any algorithm is susceptible to producing floating garbage. Specifically, one may expect that configuration *S* produces more floating garbage as it marks objects black during unsafe shape changes. The benchmark shows that, depending on the workload, configuration *S* reduces memory consumption in 7 out of 12 workloads and is neutral for the remaining 5 workloads. This is due to faster marking progress that reduces other sources of floating garbage such as objects marked black conservatively in the write barrier.

For MMU we report non-normalized comparisons with *I* to avoid showing potentially misleading improvements for smaller absolute values of MMU. We therefore report in Figure 20 the absolute values of MMU for *I*. As shown in in Figure 18 and Figure 19, all concurrent configurations (*S*, *LW*, *LP*) improve over *I*. For the 16.67ms time window the larger improvement observed is about 50% for *S* on `google-1`. For

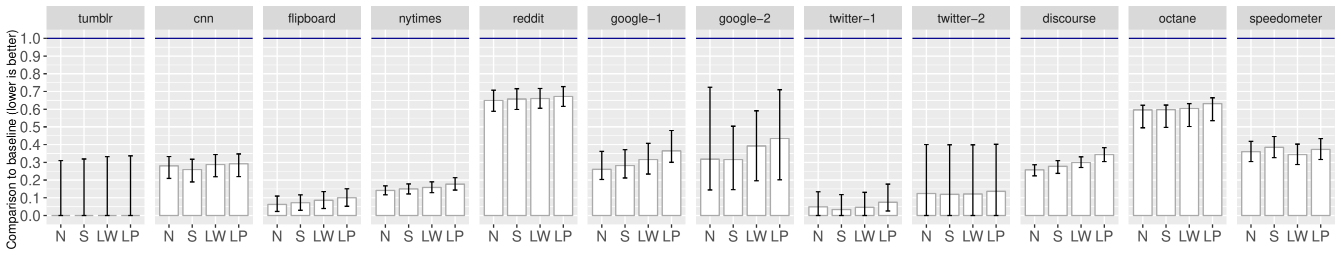


Figure 15. Cumulative marking time per garbage collection cycle on the main thread compared to the *I* baseline.

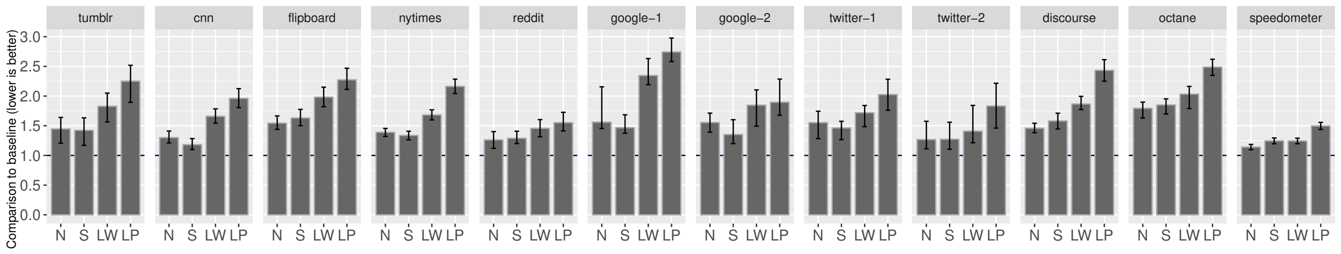


Figure 16. Cumulative marking time per garbage collection cycle on all threads compared to the *I* baseline.

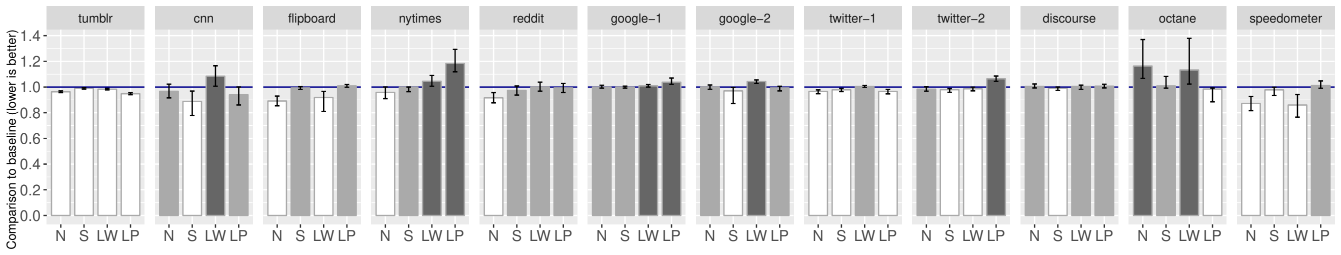


Figure 17. Average renderer memory consumption compared to the *I* baseline.

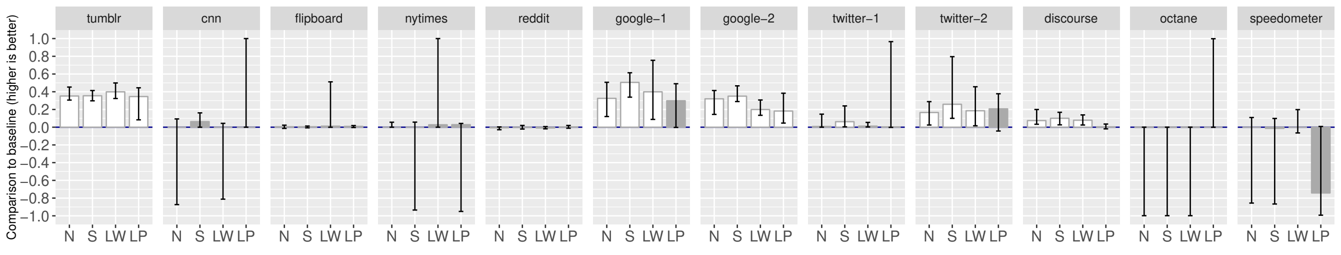


Figure 18. Difference in MMU between a configuration and the *I* baseline with 16.67ms time window.

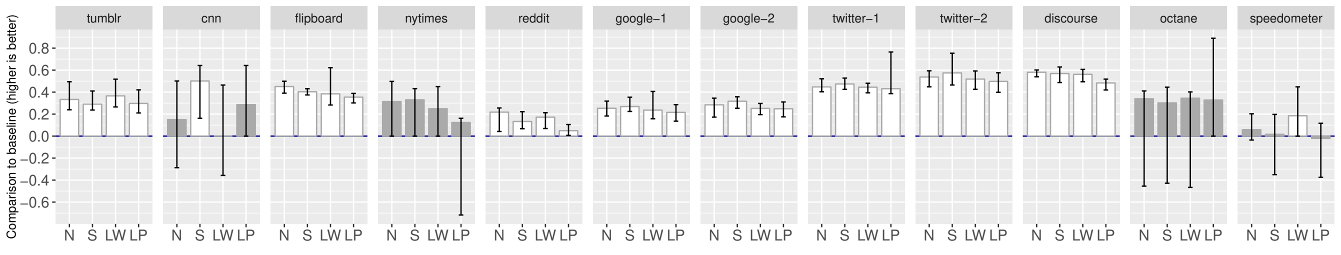
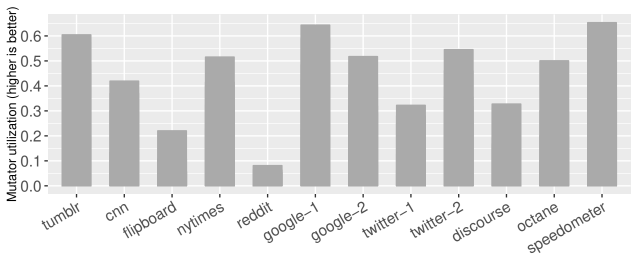
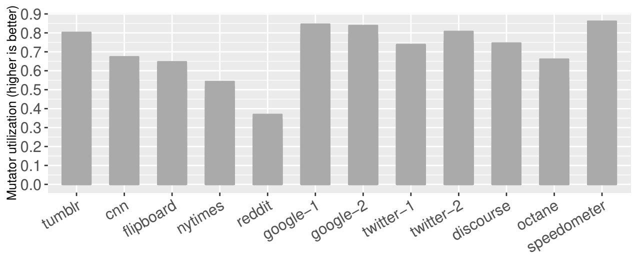


Figure 19. Difference in MMU between a configuration and the *I* baseline with 50ms time window.



(a) 16.67ms time window



(b) 50ms time window

Figure 20. MMU of the I baseline for a given time window.

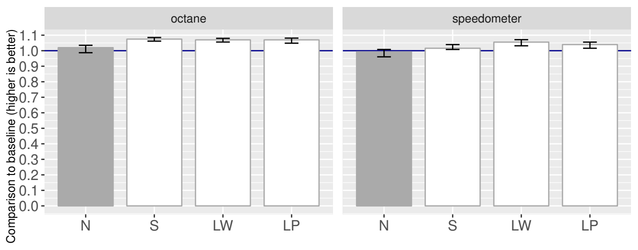


Figure 21. Total scores of Octane 2.0 and Speedometer 2.0 compared to the I baseline.

the 50ms time window all configurations provide similar improvements of up to 60% on e.g. `twitter-2`. Some pages did not improve because MMU is dominated by the main pause that also performs time-intensive tasks like compaction.

The JavaScript benchmark scores of Octane and Speedometer, are depicted in Figure 21. The total scores (averaged) for Octane are 27016 for I, 27316 for N, 28995 for S, 28857 for LW, and 28788 for LP. The snapshot-based version S shows the best performance with around 7% improvement on total score. Interestingly, I and N show similar performance. Hence plain concurrent marking and optimizations with unsafe object shape changes have similar impact on throughput in V8. Similar are the results for Speedometer, with the total scores (averaged): 126 for I, 124 for N, 129 for S, 133 for LW, and 131 for LP. The main difference is that the lock-based versions LW and LP provide the highest score followed by S. The highest scores are achieved by configurations that perform optimizations with unsafe shape changes and concurrent marking (S, LW, LP).

10 Conclusion

In this paper we discussed different object shape changing operations and classified them as safe and unsafe for concurrent marking. We showed that safe shape changes can be handled by traditional concurrent marking algorithms. To handle unsafe shape changes, we introduced a lock-based algorithm and a wait-free snapshotting algorithm, which is enabled by default in Chrome since version 64. We proved key garbage collection invariants of our algorithms and demonstrated the feasibility of our design by building it into the V8 JavaScript virtual machine where we showed performance improvements across various JavaScript benchmark suites and real-world macro-benchmarks.

This paper should make virtual machine implementers aware of object shape changes. While avoiding shape changes may result in reduced sequential performance, it enables simpler runtime and garbage collection implementations. Virtual machine implementers striving for high performance can use our wait-free snapshotting and lock-based concurrent marking algorithms to handle object shape changes efficiently with low synchronization overhead.

Acknowledgments

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

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *J. ACM* 40, 4 (Sept. 1993), 873–890. https://doi.org/10.1145/153724.153741

[2] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332

[3] Android. 2018. Android 8.0 ART Improvements. Retrieved November 3, 2018 from https://source.android.com/devices/tech/dalvik/improvements

[4] Henry G. Baker, Jr. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (April 1978), 280–294. https://doi.org/10.1145/359460.359470

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[6] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

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

[8] Perry Cheng and Guy 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

[9] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben 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/10.1145/2754169.2754181

[10] Daniel Clifford, Hannes Payer, Michael Starzinger, and Ben 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

[11] Nachshon Cohen, Arie Tal, and Erez Petrank. 2017. Layout Lock: A Scalable Locking Paradigm for Concurrent Data Layout Modifications. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 17–29. https://doi.org/10.1145/3018743.3018753

[12] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 108 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276478

[13] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes 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

[14] 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 (Oct. 2018), 24 pages. https://doi.org/10.1145/3276521

[15] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[16] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. 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

[17] MDN Web Docs. 2018. Garbage collection. Retrieved November 11, 2018 from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Garbage_collection

[18] Google. 2018. Catapult. Retrieved November 10, 2018 from https://github.com/catapult-project/catapult

[19] Google. 2018. Chrome Benchmarks. Retrieved November 10, 2018 from https://chromium.googlesource.com/chromium/src/+/master/tools/perf/benchmarks

[20] Google. 2018. Octane benchmark. Retrieved November 10, 2018 from https://chromium.github.io/octane

[21] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[22] Maurice Herlihy. 1991. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. https://doi.org/10.1145/114005.102808

[23] Myles Hollander, Douglas A. Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods.* John Wiley & Sons, Hoboken, NJ, USA.

[24] Richard L. Hudson. 2018. Getting to Go: The Journey of Go's Garbage Collector. Retrieved November 3, 2018 from https://blog.golang.org/ismmkeynote

[25] Richard L. Hudson and Austin Clements. 2016. Proposal: Eliminate STW stack re-scanning. Retrieved November 3, 2018 from https://gist.github.com/aclements/4b5e2758310032dbdb030d7648b5ab32

[26] Richard L. Hudson and J. Eliot B. Moss. 2001. Sapphire: Copying GC Without Stopping the World. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI '01)*. ACM, New York, NY, USA, 48–57. https://doi.org/10.1145/376656.376810

[27] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* CRC Press.

[28] Meggin Kearney, Addy Osmani, Kayce Basques, and Jason Miller. 2018. Measure Performance with the RAIL Model. Retrieved November 10, 2018 from https://developers.google.com/web/fundamentals/performance/rail

[29] Mozilla. 2018. Bug tracker. Retrieved November 10, 2018 from https://bugzilla.mozilla.org/show_bug.cgi?id=1364346

[30] Mozilla. 2018. Bug tracker. Retrieved November 10, 2018 from https://bugzilla.mozilla.org/show_bug.cgi?id=1348772

[31] Addy Osmani, Mathias Bynens, and Ryosuke Niwa. 2018. Speedometer 2.0. Retrieved November 10, 2018 from https://webkit.org/blog/8063/speedometer-2-0-a-benchmark-for-modern-web-app-responsiveness/

[32] Filip Pizlo. 2017. Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector. Retrieved November 11, 2018 from https://webkit.org/blog/7122

[33] Yaron Riany, Nir Shavit, and Dan Touitou. 2001. Towards a practical snapshot algorithm. *Theoretical Computer Science* 269 (2001), 163–201.

[34] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *Proceedings of the International Symposium on Memory Management (ISMM '11)*. ACM, New York, NY, USA, 79–88. https://doi.org/10.1145/1993478.1993491

[35] Taiichi Yuasa. 1990. Real-time Garbage Collection on General-purpose Machines. *Journal of Systems and Software* 11, 3 (March 1990), 181–198. https://doi.org/10.1016/0164-1212(90)90084-Y

[36] Limin Zhu. 2017. ChakraCore: Architecture Overview. Retrieved November 11, 2018 from https://github.com/Microsoft/ChakraCore/wiki/Architecture-Overview