Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector

Jan 20, 2017
by Filip Pizlo
@filpizlo

As of r209827, 64-bit ARM and x86 WebKit ports use a new garbage collector called *Riptide*. Riptide reduces worst-case pause times by allowing the app to run concurrently to the collector. This can make a big difference for responsiveness since garbage collection can easily take 10 ms or more, even on fast hardware. Riptide improves WebKit's performance on the JetStream/splay-latency test by 5x, which leads to a 5% improvement on JetStream. Riptide also improves our Octane performance. We hope that Riptide will help to reduce the severity of GC pauses for many different kinds of applications.

This post begins with a brief background about concurrent GC (garbage collection). Then it describes the Riptide algorithm in detail, including the mature WebKit GC foundation, on which it is built. The field of incremental and concurrent GC goes back a long time and WebKit is not the first system to use it, so this post has a section about how Riptide fits into the related work. This post concludes with performance data.

Introduction

Garbage collection is expensive. In the worst case, for the collector to free a single object, it needs to scan the entire heap to ensure that no objects have any references to the one it wants to free. Traditional collectors scan the entire heap periodically, and this is roughly how WebKit's collector has worked since the beginning.

The problem with this approach is that the GC pause can be long enough to cause rendering loops to miss frames, or in some cases it can even take so long as to manifest as a spin. This is a well-understood computer science problem. The originally proposed solution for janky GC pauses, by Guy Steele in 1975, was to have one CPU run the app and another CPU run the collector. This involves gnarly race conditions that Steele solved with a bunch of locks. Later algorithms like Baker's were *incremental*: they assumed that there was one CPU, and sometimes the application would call into the collector but only for bounded increments of work. Since then, a huge variety of incremental and concurrent techniques have been explored. Incremental collectors avoid some synchronization overhead, but concurrent collectors scale better. Modern concurrent collectors like DLG (short for Doligez, Leroy, Gonthier, published in POPL '93 and '94) have very cheap synchronization and almost completely avoid pausing the application. Taking garbage collection off-core rather than merely shortening the pauses is the direction we want to take in WebKit, since almost all of the devices WebKit runs on have more than one core.

The goal of WebKit's new Riptide concurrent GC is to achieve a big reduction in GC pauses by running most of the collector off the main thread. Because Riptide will be our always-on default GC, we also want it to be as efficient — in terms of speed and memory — as our previous collector.

The Riptide Algorithm

The Riptide collector combines:

- *Marking:* The collector marks objects as it finds references to them. Objects not marked are deleted. Most of the collector’s time is spent visiting objects to find references to other objects.
- *Constraints:* The collector allows the runtime to supply additional constraints on when objects should be marked, to support custom object lifetime rules.
- *Parallelism:* Marking is parallelized on up to eight logical CPUs. (We limit to eight because we have not optimized it for more CPUs.)
- *Generations:* The collector lets the mark state of objects *stick* if memory is plentiful, allowing the next collection to skip visiting those objects. *Sticky mark bits* are a common way of implementing generational collection without copying. Collection cycles that let mark bits stick are called *eden* collections in WebKit.
- *Concurrency:* Most of the collector’s marking phase runs concurrently to the program. Because this is by far the longest part of collection, the remaining pauses tend to be 1 ms or less. Riptide’s concurrency features kick in for both eden and full collections.
- *Conservatism:* The collector scans the stack and registers *conservatively*, that is, checking each word to see if it is in the bounds of some object and then marking it if it is. This means that all of the C++, assembly, and just-in-time (JIT) compiler-generated code in our system can store heap pointers in local variables without any hassles.
- *Efficiency:* This is our always-on garbage collector. It has to be fast.

This section describes how the collector works. The first part of the algorithm description focuses on the WebKit mark-sweep algorithm on which Riptide is based. Then we dive into concurrency and how Riptide manages to walk the heap while the heap is in flux.

Efficient Mark-Sweep

Riptide retains most of the basic architecture of WebKit’s mature garbage collection code. This section gives an overview of how our mark-sweep collector works: WebKit uses a *simple segregated storage* heap structure. The DOM, the Objective-C API, the type inference runtime, and the compilers all introduce *custom marking constraints*, which the GC executes to fixpoint. Marking is done in *parallel* to maximize throughput. Generational collection is important, so WebKit implements it using *sticky mark bits*. The collector uses *conservative stack scanning* to ease integration with the rest of WebKit.

Simple Segregated Storage

WebKit has long used the simple segregated storage heap structure for small and medium-sized objects (up to about 8KB):

- Small and medium-sized objects are allocated from segregated free lists. Given a desired object size, we perform a table lookup to find the appropriate free list and then pop the first object from this list. The lookup table is usually constant-folded by the compiler.
- Memory is divided into 16KB *blocks*. Each block contains *cells*. All cells in a block have the same cell size, called the block's *size class*. In WebKit jargon, an *object* is a *cell* whose JavaScript type is "object". For example, a string is a cell but not an object. The GC literature would typically use *object* to refer to what our code would call a *cell*. Since this post is not really concerned with JavaScript types, we'll use the term *object* to mean any cell in our heap.
- At any time, the active free list for a size class contains only objects from a single block. When we run out of objects in a free list, we find the next block in that size class and *sweep* it to give it a free list.

Sweeping is incremental in the sense that we only sweep a block just before allocating in it. In WebKit, we optimize sweeping further with a hybrid bump-pointer/free-list allocator we call *bump'n'pop* (here it is in C++ and in the compilers). A per-block bit tells the sweeper if the block is completely empty. If it is, the sweeper will set up a bump-pointer arena over the whole block rather than constructing a free-list. Bump-pointer arenas can be set up in O(1) time while building a free-list is a O(n) operation. Bump'n'pop achieves a big speed-up on programs that allocate a lot because it avoids the sweep for totally-empty blocks. Bump'n'pop's bump-allocator always bumps by the block's cell size to make it look like the objects had been allocated from the free list. This preserves the block's membership in its size class.

Large objects (larger than about 8KB) are allocated using malloc.

Constraint-Based Marking

Garbage collection is ordinarily a graph search problem and the heap is ordinarily just a graph: the roots are the local variables, their values are directional edges that point to objects, and those objects have fields that each create edges to some other objects. WebKit's garbage collector also allows the DOM, compiler, and type inference system to install constraint callbacks. These constraints are allowed to query which objects are marked and they are allowed to mark objects. The WebKit GC algorithm executes these constraints to fixpoint. GC termination happens when all marked objects have been visited and none of the constraints want to mark anymore objects. In practice, the constraint-solving part of the fixpoint takes up a tiny fraction of the total time. Most of the time in GC is spent performing a depth-first search over marked objects that we call *draining*.

Parallel Draining

Draining takes up most of the collector's time. One of our oldest collector optimizations is that draining is parallelized. The collector has a draining thread on each CPU. Each draining thread has its own worklist of objects to visit, and ordinarily it runs a graph search algorithm that only sees this worklist. Using a local worklist means avoiding worklist synchronization most of the time. Each draining thread will check in with a global worklist under these conditions:

- It runs out of work. When a thread runs out of work, it will try to steal 1/Nth of the global worklist where N is the

number of idle draining threads. This means acquiring the global worklist's lock.
- Every 100 objects visited, the draining thread will consider donating about half of its worklist to the global worklist. It will only do this if the global worklist is empty, the global worklist lock can be acquired without blocking, and the local worklist has at least two entries.

This algorithm appears to scale nicely to about eight cores, which is good enough for the kinds of systems that WebKit usually runs on.

Draining in parallel means having to synchronize marking. Our marking algorithm uses a lock-free CAS (atomic compare-and-swap instruction) loop to set mark bits.

Sticky Mark Bits

Generational garbage collection is a classic throughput optimization first introduced by Lieberman and Hewitt and Ungar. It assumes that objects that are allocated recently are unlikely to survive. Therefore, focusing the collector on objects that were allocated since the last GC is likely to free up lots of memory — almost as much as if we collected the whole heap. Generational collectors track the *generation* of objects: either young or old. Generational collectors have (at least) two modes: *eden* collection that only collects young objects and *full* collection that collects all objects. During an eden collection, old objects are only visited if they are suspected to contain pointers to new objects.

Generational collectors need to overcome two hurdles: how to track the generation of objects, and how to figure out which old objects have pointers to new objects.

The collector needs to know the generation of objects in order to determine which objects can be safely ignored during marking. In a traditional generational collector, eden collections move objects and then use the object's address to determine its generation. Our collector does not move objects. Instead, it uses the mark bit to also track generation. Quite simply, we don't clear any mark bits at the start of an eden collection. The marking algorithm will already ignore objects that have their mark bits set. This is called *sticky mark bit* generational garbage collection.

The collector will avoid visiting old objects during an eden collection. But it cannot avoid all of them: if an old object has pointers to new objects, then the collector needs to know to visit that old object. We use a *write barrier* — a small piece of instrumentation that executes after every write to an object — that tells the GC about writes to old objects. In order to cheaply know which objects are old, the object header also has a copy of the object's state: either it is old or it is new. Objects are allocated new and labeled old when marked. When the write barrier detects a write to an old object, we tell the GC by setting the object's state to *old-but-remembered* and putting it on the mark stack. We use separate mark stacks for objects marked by the write barrier, so when we visit the object, we know whether we are visiting it due to the barrier or because of normal marking (i.e. for the first time). Some accounting only needs to happen when visiting the object for the first time. The complete barrier is simply:

```
object->field = newValue;
if (object->cellState == Old)
    remember(object);
```

Generational garbage collection is an enormous improvement in performance on programs that allocate a lot, which is common in JavaScript. Many new JavaScript features, like iterators, arrow functions, spread, and for-of allocate lots of objects and these objects die almost immediately. Generational GC means that our collector does not need to visit all of the old objects just to delete the short-lived garbage.

Conservative Roots

Garbage collection begins by looking at local variables and some global state to figure out the initial set of marked objects. Introspecting the values of local variables is tricky. WebKit uses C++ local variables for pointers to the garbage collector's heap, but C-like languages provide no facility for precisely introspecting the values of specific variables of arbitrary stack frames. WebKit solves this problem by marking objects conservatively when scanning roots. We use the simple segregated storage heap structure in part because it makes it easy to ask whether an arbitrary bit pattern could possibly be a pointer to some object.

We view this as an important optimization. Without conservative root scanning, C++ code would have to use some API to notify the collector about what objects it points to. Conservative root scanning means not having to do any of that work.

Mark-Sweep Summary

Riptide implements complex notions of reachability via arbitrary constraint callbacks and allows C++ code to manipulate objects directly. For performance, it parallelizes marking and uses generations to reduce the average amount of marking work.

Handling Concurrency

Riptide makes the draining phase of garbage collection concurrent. This works because of a combination of concurrency features:

- Riptide is able to *stop the world* for certain tricky operations like stack scanning and DOM constraint solving.
- Riptide uses a *retreating wavefront* write barrier to manage races between marking and object mutation. Using retreating wavefront allows us to avoid any impedance mismatch between generational and concurrent collector optimizations.
- Retreating wavefront collectors can suffer from the risk of GC death spirals, so Riptide uses a *space-time scheduler* to put that in check.
- Visiting an object while it is being reshaped is particularly hard, and WebKit reshapes objects as part

of type inference. We use an *obstruction-free double collect snapshot* to ensure that the collector never marks garbage memory due to a visit-reshape race.
- Lots of objects have tricky races that aren't on the critial path, so we put a fast, adaptive, and fair lock in every JavaScript object as a handy way to manage them. It fits in two otherwise unused bits.

While we wrote Riptide for WebKit, we suspect that the underlying intuitions could be useful for anyone wanting to write a concurrent, generational, parallel, conservative, and non-copying collector. This section describes Riptide in detail.

Stopping The World and Safepoints

Riptide does draining concurrently. It is a goal to eventually make other phases of the collector concurrent as well. But so long as some phases are not safe to run concurrently, we need to be able to bring the application to a stop before performing those phases. The place where the collector stops needs to be picked so as to avoid reentrancy issues: for example stopping to run the GC in the middle of the GC's allocator would create subtle problems. The concurrent GC avoids these problems by only stopping the application at those points where the application would trigger a GC. We call these *safepoints*. When the collector brings the application to a safepoint, we say that it is *stopping the world*.

Riptide currently stops the world for most of the constraint fixpoint, and resumes the world for draining. After draining finishes, the world is again stopped. A typical collection cycle may have many stop-resume cycles.

Retreating Wavefront

Draining concurrently means that just as we finish visiting some object, the application may store to one of its fields. We could store a pointer to an unmarked object into an object that is already visited, in which case the collector might never find that unmarked object. If we don't do something about this, the collector would be sure to prematurely delete objects due to races with the application. Concurrent garbage collectors avoid this problem using write barriers. This section describes Riptide's write barrier.

Write barriers ensure that the state of the collector is still valid after any race, either by marking objects or by having objects revisited (GC Handbook, chapter 15). Marking objects helps the collector make forward progress; intuitively, it is like *advancing* the collector's *wavefront*. Having objects revisited *retreats* the wavefront. The literature of full of concurrent GC algorithms, like the Metronome, C4, and DLG, that all use some kind of advancing wavefront write barrier. The simplest such barrier is Dijkstra's, which marks objects anytime a reference to them is created. I used these kinds of barriers in my past work because they make it easy to make the collector very deterministic. Adding one of those barriers to WebKit would be likely to create some performance overhead since this means adding new code to every write to the heap. But the retreating wavefront barrier, originally invented by Guy Steele in 1975, works on exactly the same principle as our existing generational barrier. This allows Riptide to achieve zero barrier overhead by reusing WebKit's existing barrier.

It's easiest to appreciate the similarity by looking at some barrier code. Our old generational barrier looked like this:

```
object->field = newValue;
if (object->cellState == Old)
    remember(object);
```

Steele's retreating wavefront barrier looks like this:

```
object->field = newValue;
if (object->cellState == Black)
    revisit(object);
```

Retreating wavefront barriers operate on the same principle as generational barriers, so it's possible to use the same barrier for both. The only difference is the terminology. The *black* state means that the collector has already visited the object. This barrier tells the collector to revisit the object if its `cellState` tells us that the collector had already visited it. This state is part of the classic *tri-color abstraction*: *white* means that the GC hasn't marked the object, *grey* means that the object is marked and on the mark stack, and *black* means that the object is marked and has been visited (so is not on the mark stack anymore). In Riptide, the tri-color states that are relevant to concurrency (white, grey, black) perfectly overlap with the sticky mark-bit states that are relevant to generations (new, remembered, old). The Riptide cell states are as follows:

- DefinitelyWhite: the object is new and white.
- PossiblyGrey: the object is grey, or remembered, or new and white.
- PossiblyBlack: the object is black and old, or grey, or remembered, or new and white.

A naive combination generational/concurrent barrier might look like this:

```
object->field = newValue;
if (object->cellState == PossiblyBlack)
    slowPath(object);
```

This turns out to need tweaking to work. The `PossiblyBlack` state is too ambiguous, so the `slowPath` needs additional logic to work out what the object's state really was. Also, the order of execution matters: the CPU must run the `object->cellState` load after it runs the `object->field` store. That's hard, since CPUs don't like to obey store-before-load orderings. Finally, we need to guarantee that the barrier cannot retreat the wavefront too much.

Disambiguating Object State

The GC uses the combination of the object's mark bit in the block header and the `cellState` byte in the object's header to determine the object's state. The GC clears mark bits at the start of full collection, and it sets the `cellState` during marking and barriers. It doesn't reset objects' cellStates back to `DefinitelyWhite` at the start of a full collection, because it's possible to infer that the cellState should have been reset by looking at the mark bit. It's important that the collector never scans the heap to clear marking state, and even mark bits are logically cleared using versioning. If an object is PossiblyBlack or PossiblyGrey and its mark bit is logically clear, then this means that the object is really white. Riptide's barrier `slowPath` is almost like our old generational slow path but it has a new check: it will not do anything if the mark bit of the target object is not set, since this means that we're in the middle of a GC and the object is actually white. Additionally, the barrier will attempt to set the object back to `DefinitelyWhite` so that the `slowPath` path does not have to see the object again (at least not until it's marked and visited).

Store-Before-Barrier Ordering

The GC must flag the object as PossiblyBlack just before it starts to visit it and the application must store to `field` before loading `object->cellState`. Such ordering is not guaranteed on any modern architecture: both x86 and ARM will sink the store below the load in some cases. Inserting an unconditional store-load fence, such as `lock; orl $0, (%rsp)` on x86 or `dmb ish` on ARM, would degrade performance way too much. So, we make the fence itself conditional by playing a trick with the barrier's condition:

```
object->field = newValue;
if (object->cellState <= blackThreshold)
    slowPath(object);
```

Where `blackThreshold` is a global variable. The `PossiblyBlack` state has the value 0, and when the collector is not running, `blackThreshold` is 0. But once the collector starts marking, it sets `blackThreshold` to 100 while the world is stopped. Then the barrier's `slowPath` leads with a check like this:

```
storeLoadFence();
if (object->cellState != PossiblyBlack)
    return;
```

This means that the application takes a slight performance hit while Riptide is running. In typical programs, this overhead is about 5% during GC and 0% when not GCing. The only additional cost when not GCing is that `blackThreshold` must be loaded from memory, but we could not detect a slow-down due to this change. The 5% hit during collection is worth fixing, but to put it in perspective, the application used to take a 100% performance hit during GC because the GC would stop the application from running.

The complete Riptide write barrier is emitted as if the following `writeBarrier` function had been inlined just after any store to `target`:

```
ALWAYS_INLINE void writeBarrier(JSCell* target)
{
    if (LIKELY(target->cellState() > blackThreshold)
        return;
    storeLoadFence();
    if (target->cellState() != PossiblyBlack)
        return;
    writeBarrierSlow(target);
}

NEVER_INLINE void writeBarrierSlow(JSCell* target)
{
    if (!isMarked(target)) {
        // Try to label this object white so that we
```

```
            // slow path again.
            if (target->compareExchangeCellState(Possibl
                if (Heap::isMarked(target)) {
                    // A race! The GC marked the object
                    // pessimistically label it black ag
                    target->setCellState(PossiblyBlack);
                }
            }
            return;
        }

        target->setCellState(DefinitelyGrey);
        m_mutatorMarkStack->append(target);
    }
```

The JIT compiler inlines the part of the slow path that rechecks the object's state after doing a fence, since this helps keep the overhead low during GC. Moreover, our just-in-time compilers optimize the barrier further by removing barriers if storing values that the GC doesn't care about, removing barriers on newly allocated objects (which must be white), clustering barriers together to amortize the cost of the fence, and removing redundant barriers if an object is stored to repeatedly.

Revisiting

When the barrier does append the object to the `m_mutatorMarkStack`, the object will get revisited eventually. The revisit could happen concurrently to the application. That's important since we have seen programs retreat the wavefront enough that the total revisit pause would be too big otherwise.

Unlike advancing wavefront, retreating wavefront means forcing the collector to redo work that it has already done. Without some facilities to ensure collector progress, the collector might never finish due to repeated revisit requests from the write barrier. Riptide tackles this problem in two ways. First, we defer all revisit requests. Draining threads do not service any revisit requests until they have no other work to do. When an object is flagged for revisiting, it stays in the *grey* state for a while and will only be revisited towards the end of GC. This ensures that if an old object often has its fields overwritten with pointers to new objects, then the GC will usually only scan two snapshots' worth of those fields: one snapshot whenever the GC visited the object first, and another towards the end when the GC gets around to servicing deferred revisits. Revisit deferral reduces the likelihood of runaway GC, but fully eliminating such pathologies is left to our scheduler.

Space-Time Scheduler

The bitter end of a retreating wavefront GC cycle is not pretty: just as the collector goes to visit the last object on the mark stack, some object that had already been visited gets written to, and winds up back on the mark stack. This can go on for a while, and before we had any mitigations we saw Riptide using 5x more memory than with synchronous collection. This death spiral happens because programs allocate a lot all the time and the collector cannot free any memory until it finishes marking. Riptide prevents death spirals using a scheduler that controls the application's pace. We call it the *space-time* scheduler because it links the amount of *time* that the application gets to run for in a timeslice to the amount of *space* that the application has used by allocating in the collector's *headroom*.

The space-time scheduler ensures that the retreating wavefront barrier cannot wreak havoc by giving the collector an unfair advantage: it will periodically stop the world for short pauses even when the collector could be running concurrently. It does this just so the collector can always outpace the application in case of a race. If this was meant as a garbage collector for servers, you could imagine providing the user with a bunch of knobs to control the schedule of these synthetic pauses. Different applications will have different ideal pause lengths. Applications that often write to old memory will retreat the collector's wavefront a lot, and so they will need a longer pause to ensure termination. Functional-style programs tend to only write to newly allocated objects, so those could get away with a shorter pause. We don't want web users or web developers to have to configure our collector, so the space-time scheduler adaptively selects a pause schedule.

To be correct, the scheduler must eventually pause the world for long enough to let the collector terminate. The space-time scheduler is based on a simple idea: the length of pauses increases during collection in response to how much memory the application is using.

The space-time scheduler selects the duration and spacing of synthetic pauses based on the *headroom ratio*, which is a measure of the amount of extra memory that the application has allocated during the concurrent collection. A concurrent collection is triggered by memory usage crossing the *trigger threshold*. Since the collector allows the application to keep running, the application will keep allocating. The space that the collector makes available for allocation during collection is called the *headroom*. Riptide is tuned for a *max headroom* that is 50% larger than the trigger threshold: so if the app needed to allocate 100MB to trigger a collection, its max headroom is 50MB. We want the collector to complete synchronously if we ever deplete all of our headroom: at that point it's better for the system to pause and free memory than to run and deplete even more memory. The headroom ratio is simply the available headroom divided by the max headroom. The space-time scheduler will divide time into fixed timeslices, and the headroom ratio determines how much time the application is resumed for during that period.

The default tuning of our collector is that the collector timeslice is 2 ms, and the first *C* ms of it is given to the collector and the remaining *M* ms is given to the mutator. We always let the collector pause for at least 0.6 ms. Let *H* be the headroom ratio: 1 at the start of collection, and 0 if we deplete all headroom. With a 0.6 ms minimum pause and a 2 ms timeslice, we define *M* and *C* as follows:

$$M = 1.4\ \mathrm{H}$$
$$C = 2 - \mathrm{M}$$

For example, at the start of usual collection we will give 0.6 ms to the collector and then 1.4 ms to the application, but as soon as the application starts allocating, this window shifts. Aggressive applications, which both allocate a lot and write to old objects a lot, will usually end collection with the split being closer to 1 ms for the collector followed by 1 ms for the application.

Thanks to the space-time scheduler, the worst that an adversarial program could do is cause the GC to keep revisiting some object. But it can't cause the GC to run out of memory, since if the adversary uses up all of the headroom then *M* becomes 0 and the collector gets to stop the world until the end of the cycle.

Obstruction-Free Double Collect Snapshot

Concurrent garbage collection means finding exciting new ways of side-stepping expensive synchronization. In traditional concurrent mark-sweep GCs, which focused on nicely-typed languages, the worst race was the one covered by the write barrier. But since this is JavaScript, we get to have a lot more fun.

JavaScript objects may have properties added to them at any time. The WebKit JavaScript object model has three features that makes this efficient:

- Each object has a *structure ID*: The first 32 bits of each object is its structure ID. Using a table lookup, this gives a pointer to the object's *structure*: a kind of meta-object that describes how its object is supposed to look. The object's layout is governed by its structure. Some objects have immutable structures, so for those we know that so long as their structure IDs stay the same, they will be laid out the same.
- The structure may tell us that the object has *inline storage*. This is a slab of space in the object itself, left aside for JavaScript properties.
- The structure may tell us about the object's *butterfly*. Each object has room for a pointer that can be used to point to an overflow storage for additional properties

that we call a butterfly. The butterfly is a bidirectional object that may store named properties to the left of the pointer and indexed properties to the right.

It's imperative that the garbage collector visits the butterfly using exactly the structure that corresponds to it. If the object has a mutable structure, it's imperative that the collector visits the butterfly using the data from the structure that corresponds to that butterfly. The collector would crash if it tried to decode the butterfly using wrong information.

To accomplish this, we use a very simple obstruction-free version of Afek et al's double collect snapshot. To handle the immutable structure case, we just ensure that the application uses this protocol to set both the structure and butterfly:

1. Nuke the structure ID — *this sets a bit in the structure ID to indicate to the GC that the structure and butterfly are changing*.
2. Set the butterfly.
3. Set the new (decontaminated) structure ID — *decontaminating means clearing the nuke bit*.

Meanwhile the collector does this to read both the structure and the butterfly:

1. Read the structure ID.
2. Read the butterfly.
3. Read the structure ID again, and compare to (1).

If the collector ever reads a nuked structure ID, or if the structure ID in (1) and (3) are different, then we know that we will have a butterfly-structure mismatch. But if none of these conditions hold, then we are guaranteed that the collector will have a consistent structure and butterfly. See here for the proof.

Harder still is the case where the structure is mutable. In this case, we ensure that the protocol for setting the fields in the structure is to set them after the structure is nuked but before the new one is installed. The collector reads those fields before/after as well. This allows the collector to see a consistent snapshot of the structure, butterfly, and a bit inside the structure without using any locking. All that matters is that the stores in the application and the loads in the collector are ordered. We get this for free on x86, and on ARM we use store-store fences in the application (`dmb ishst`) and load-load fences in the collector (`dmb ish`).

This algorithm is said to be obstruction-free because it will complete in O(1) time no matter what kind of race it encounters, but if it does encounter a race then it'll tell you to try again. Obstruction-free algorithms need some kind of *contention manager* to ensure that they do eventually complete. The contention manager must provably maximize the likelihood that the obstruction-free algorithm will eventually run without any race. For example, this would be a sound contention manager: exponential back-off in which the actual back-off amount is a random number between 0 and X where X increases exponentially on each try. It turns out that Riptide's retreating wavefront revisit scheduler is already a natural contention manager. When the collector bails on visiting an object because it detected a race, it schedules revisiting of that object just as if a barrier had executed. So, the GC will visit any object that encountered such a race again anyway. The GC will visit the object much later and the timing will be somewhat pseudo-random due to OS scheduling. If an object did keep getting revisited, eventually the space-time scheduler will increase the collector's synthetic pause to the point where the revisit will happen with the world stopped. Since there are no safepoints possible in any of the structure/butterfly atomic protocols, stopping the world ensures that the algorithm will not be obstructed.

Embedded WTF Locks

The obstruction-free object snapshot is great, but it's not scalable — from a WebKit developer sanity standpoint — to use it everywhere. Because we have been adding more concurrency to WebKit for a while, we made this easier by already having a custom locking infrastructure in WTF (Web

Template Framework). One of the goals of WTF locks was to fit locks in two bits so that we may one day stuff a lock into the header of each JavaScript object. Many of the loony corner-case race conditions in the concurrent garbage collector happen on paths where acquiring a lock is fine, particularly if that lock has a great inline fast path like WTF locks. So, all JavaScript objects in WebKit now have a fast, adaptive, and fair WTF lock embedded in two bits of what is otherwise the *indexingType* byte in the object header. This *internal lock* is used to protect mutations to all sorts of miscellaneous data structures. The collector will hold the internal lock while visiting those objects.

Locking should always be used with care since it can be a slow-down. In Riptide, we only use locking to protect uncommon operations. Additionally, we use an optimized lock implementation to reduce the cost of synchronization even further.

Algorithm Summary

Riptide is an improvement to WebKit's collector and retains most of the things that made the old algorithm great. The changes that transformed WebKit's collector were landed over the past six months, starting with the painful work of removing WebKit's previous use of copying. Riptide combines Guy Steele's classic retreating wavefront write barrier with a mature sticky-mark-sweep collector and lots of concurrency tricks to get a useful combination of high GC throughput and low GC latency.

Related Work

The paper that introduced retreating wavefront did not claim to implement the idea — it was just a thought experiment. We are aware of two other implementations of retreating wavefront. The oldest is the BDW (Boehm-Demers-Weiser) collector's incremental mode. That collector uses a page-granularity revisit because it relies entirely on page faults to trigger the barrier. The collector makes pages that have black objects read-only and then any write to that page triggers a fault. The fault handler makes the page read-write and logs the *entire* page for revisiting. Riptide uses a software barrier that precisely triggers revisiting only for the object that got stored to. The BDW collector uses page faults for a good reason: so that it can be used as a plug-in component to any kind of language environment. The compiler doesn't have to be aware of retreating wavefronts or generations since the BDW collector will be sure to catch all of the writes that it cares about. But in WebKit we are happy to have everything tightly integrated and so Riptide relies on the rest of WebKit to use its barrier. This was not hard since the new barrier is almost identical to our old one.

Another user of retreating wavefront is ChakraCore. It appears to have both a page-fault-based barrier like BDW and a software card-marking barrier that can flag 128-byte regions of memory as needing revisit. (For a good explanation of card-marking, albeit in a different VM, see here.) Riptide uses an object-granularity barrier instead. We tried card-marking, but found that it was slower than our barrier unless we were willing to place our entire heap in a single large virtual memory reservation. We didn't want our memory structure to be that deterministic. All retreating wavefront collectors require a stop-the-world snapshot-at-the-end increment that confirms that there is no more marking left to do. Both BDW and ChakraCore perform all revisiting during the snapshot-at-the-end. If there is a lot of revisiting work, that increment could take a while. That risk is particularly high with card-marking or fault-based barriers, in which a write to a single object usually causes the revisiting of multiple objects. Riptide can revisit objects with the application resumed. Riptide can also resume the application in between executions of custom constraints. Riptide is tuned so that the snapshot-at-the-end is only confirming that there is no more work, rather than spending an unbounded amount of time creating and chasing down new work.

Instead of retreating wavefront, most incremental, concurrent, and real-time collectors use some kind of advancing wavefront barrier. In those kinds of barriers, the application marks the objects it interacts with under certain conditions. Baker's barrier marks every pointer you load from the heap. Dijkstra's barrier marks every pointer you store into the heap. Yuasa's barrier marks every pointer you overwrite. All of these barriers *advance* the collector's wavefront in the sense that they reduce the amount of work that the collector will have to do — the thinking goes that the collector would have marked the object anyway so the barrier is helping. Since these collectors usually allocate objects black during collection, marking objects will not postpone when the collector can finish. This means that advancing wavefront collectors will mark all objects that were live at the very beginning of the cycle and all objects allocated during the cycle. Keeping objects allocated during the GC cycle (which may be long) is called *floating garbage*. Retreating wavefront collectors largely avoid floating garbage since in those collectors an object can only be marked if it is found to be referenced from another marked object.

Advancing wavefront barriers are not a great match for generational collection. The generational barrier isn't going to overlap with an advancing wavefront barrier the way that Riptide's, ChakraCore's, and BDW's do. This means double the barrier costs. Also, in an advancing wavefront generational collector, eden collections have to be careful to ensure that their floating garbage doesn't get promoted. This requires distinguishing between an object being marked for survival versus being marked for promotion. For example, the Domani, Kolodner, Petrank collector has a "yellow" object state and special color-toggling machinery to manage this state, all so that it does not promote floating garbage. The Frampton, Bacon, Cheng, and Grove version of the Metronome collector maintains three nurseries to gracefully move objects between generations, and in their collector the eden collections and full collections can proceed concurrently to each other. While those collectors have incredible features, they are not in widespread use, probably because of increased baseline costs due to extra bookkeeping and extra barriers. To put in perspective how annoying the concurrent-generational integration is, many systems like V8 and HotSpot avoid the problem by using synchronous eden collections. We want eden collections to be concurrent because although they are usually fast, we have no bound on how long they could take in the worst case. Not having floating garbage is another reason why it's so easy for retreating wavefront collectors to do concurrent eden collection: there's no need to invent states for black-but-new objects.

Using retreating wavefront means we don't get the advancing wavefront's GC termination guarantee. We make up for it by having more aggressive scheduling. It's common for advancing wavefront collectors to avoid all global pauses because all of collection is concurrent. In the most aggressive advancing wavefront concurrent collectors, the closest thing to a "pause" is that at some point each thread must produce a stack scan. Even if all of Riptide's algorithms were concurrent, we would still have to artificially stop the application simply to ensure termination. That's a trade-off that we're happy with, since we get to control how long these synthetic pauses are.

In many ways, Riptide is a classic mark-sweep collector. Using simple segregated storage is very common, and variants of this technique can be found in Jikes RVM, the Metronome real-time garbage collector, the BDW collector, the Bartok concurrent mark-sweep collector, and probably many others. Combining mark-sweep with bump-pointer is not new; Immix is another way to do it. Our bump'n'pop allocator looks most like Hoard's, and the technique was also used in Vam and reaps. Our conservative scan is almost like what the BDW collector does. Sticky mark bits are also used in BDW, Jikes RVM, and ChakraCore.

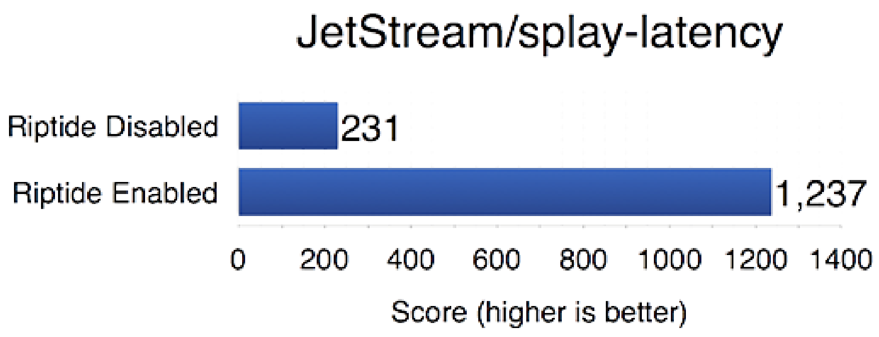
Evaluation

We enabled Riptide once we were satisfied that it did not have any major remaining regressions (in stability, performance, and memory usage) and that it demonstrated an improvement on some test of GC pauses. Enabling it now enables us to expose it to a lot of testing as we continue to tune and validate this collector. This section summarizes what we know about Riptide's performance so far.

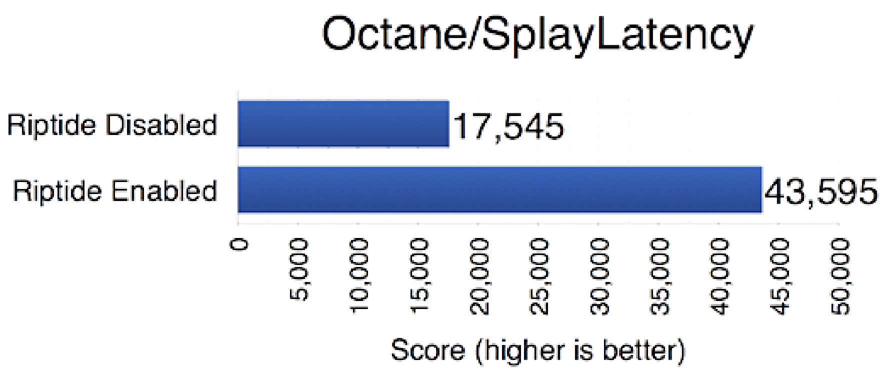
The synchronization features that enable concurrent collection were landed in many revisions over a six month period starting in July 2016. This section focuses on the performance boost that we get once we enable Riptide. Enabling Riptide means that draining will resume the application and allow the application and collector to run alongside each other. The application will still experience pauses: both synthetic pauses from the space-time scheduler and mandatory pauses for things like DOM constraint evaluation. The goal of this evaluation is to give a glimpse of what Riptide can do for observed pauses.

The test that did the best job of demonstrating our garbage collector's jankyness was the Octane SplayLatency test. This test is also included in JetStream. WebKit was previously not the best at either version of this test so we wanted a GC that would give us a big improvement. The Octane version of this test reports the reciprocal of the root-mean-squared, which rewards uniform performance. JetStream reports the reciprocal of the average of the worst 0.5% of samples, which rewards fast worst-case performance. We tuned Riptide on the JetStream version of this test, but we show results from both versions.

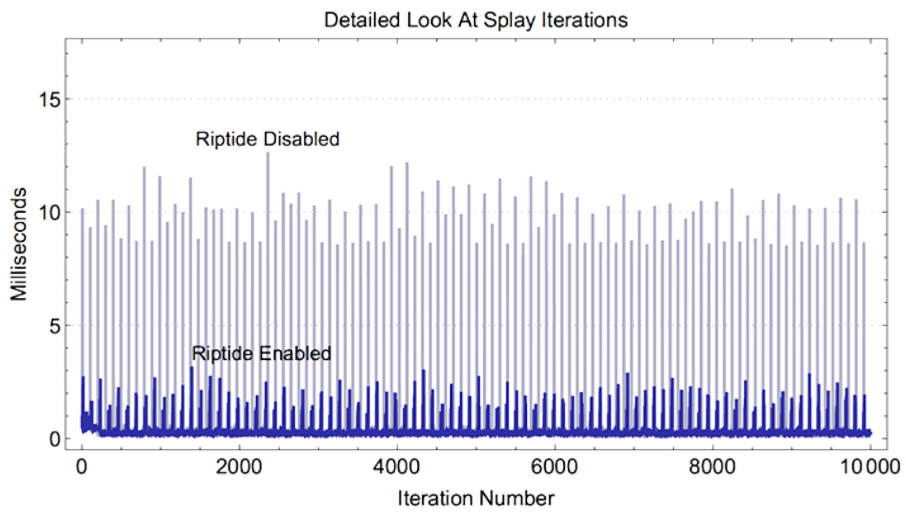
The performance data was gathered on a 15" MacBook Pro with a 2.8 GHz Intel Core i7 and 16GB RAM. This machine has four cores, and eight logical CPUs thanks to hyperthreading. We took care to quiet down the machine before running benchmarks, by closing almost all apps, disconnecting from the network, disabling Spotlight, and disabling ReportCrash. Our GC is great at taking advantage of hyperthreaded CPUs, so it runs eight draining threads on this machine.



The figure above shows that Riptide improves the JetStream/splay-latency score by a factor of five.



The figure above shows that Riptide improves the Octane/SplayLatency score by a factor of 2.5.



The chart above shows what is happening over 10,000 iterations of the Splay benchmark: without Riptide, an occasional iteration will pause for >10 ms due to garbage collection. Enabling Riptide brings these hiccups below 3 ms.

You can run this benchmark interactively if you want to see how your browser's GC performs. That version will plot the time per iteration in milliseconds over 2,000 iterations.

We continue to tune Riptide as we validate it on a larger variety of workloads. Our goal is to continue to reduce pause times. That means making more of the collector concurrent and improving the space-time scheduler. Continued tuning is tracked by bug 165909.

Conclusion

This post describes the new Riptide garbage collector in WebKit. Riptide does most of its work off the main thread, allowing for a significant reduction in worst-case pause times. Enabling Riptide leads to a five-fold improvement in latency as reported by the JetStream/splay-latency test. Riptide is now enabled by default in WebKit trunk and you can try it out in Safari Technology Preview 21. Please try it out and file bugs! ■

Next

Release Notes for Safari Technology Preview 22

Learn more

Previously

Release Notes for Safari Technology Preview 21

Learn more