Understanding Garbage Collection in JavaScriptCore From Scratch

JavaScript relies on garbage collection (GC) to reclaim memory. In this post, we will dig into JSC's garbage collection system.

Before we start, let me briefly introduce myself. I am Haoran Xu, a PhD student at Stanford University. While I have not yet contributed a lot to JSC, I found JSC a treasure of elegant compiler designs and efficient implementations, and my research is exploring ways to transfer JSC's design to support other programming languages like Lua at a low engineering cost. This post was initially posted on my blog — great thanks to the WebKit project for cross-posting it on their official blog!

Filip Pizlo's blog post on GC is great at explaining the novelties of JSC's GC, and also positions it within the context of various GC schemes in academia and industry. However, as someone with little GC background, I felt the blog alone insufficient for me to get a solid understanding of the algorithm and the motivation behind the design. Through digging into the code, and with some great help from Saam Barati, one of JSC's lead developers, I wrote up this blog post in the hope that it can help more people understand this beautiful design.

The garbage collector in JSC is non-compacting, generational and mostly[1]–concurrent. On top of being concurrent, JSC's GC heavily employs lock-free programming for better performance.

As you can imagine, JSC's GC design is quite complex. Instead of diving into the complex invariants and protocols, we will start with a simple design, and improve it step by step to converge at JSC's design. This way, we not only understand *why* JSC's design works, but also *how* JSC's design was constructed over time.

But first of all, let's get into some background.

Memory Allocation in JSC

Memory allocators and GCs are tightly coupled by nature – the allocator allocates memory to be reclaimed by the GC, and the GC frees memory to be reused by the allocator. In this section, we will briefly introduce JSC's memory allocators.

At the core of the memory allocation scheme in JSC is the data structure BlockDirectory[2]. It implements a fixed-sized allocator, that is, an allocator that only allocates memory chunks of some fixed size `s`. The allocator keeps tracks of a list of fixed-sized (in current code, 16KB) memory pages ("blocks") it owns, and a free list. Each block is divided into cells of size `s`, and has a footer at its end[3], which contains metadata needed for the GC and allocation, e.g., which cells are free. By aggregating and sharing metadata at the footer, it both saves memory and improves performance of related operations: we will go into the details later in this post.

When a `BlockDirectory` needs to make an allocation, it tries to allocate from its free list. If the free list is empty, it tries to iterate through the blocks it owns[4], to see if it can find a block containing free cells (which are marked free by GC). If yes, it scans the block footer metadata to find out all the free cells[5] in this block, and put into the free list. Otherwise, it allocates a new block from malloc[6]. Note

that this implies a `BlockDirectory`'s free list only contains cells in one block: this is called `m_currentBlock` in the code, and we will revisit this later.

`BlockDirectory` is used as the building block to build the memory allocators in JSC. JSC employs three kinds of allocators:

1. CompleteSubspace: this is a segregated allocator responsible for allocating small objects (max size about 8KB). Specifically, there is a pre-defined list of exponentially-growing size-classes[7], and one `BlockDirectory` is used to handle allocation for each size class. So to allocate an object, you find the smallest size class large enough to hold the object, and allocate from the directory for that size class.
2. PreciseAllocation: this is used to handle large allocations that cannot be handled by the `CompleteSubspace` allocator[8]. It simply relies on the standard (malloc-like) memory allocator, though in JSC a custom malloc implementation called `libpas` is used. The downside is that since a `PreciseAllocation` is created on a per-object basis, the GC cannot aggregate and share metadata information of multiple objects together to save memory and improve performance (as `MarkedBlock`'s block footer did). Therefore, every `PreciseAllocation` comes with a whopping overhead of a 96-byte GC header to store the various metadata information needed for GC for this object (though this overhead is justified since each allocation is already at least 8KB).
3. IsoSubspace: each `IsoSubspace` is used to allocate objects of a fixed type with a fixed size. So each `IsoSubspace` simply holds a `BlockDirectory` to do allocation (though JSC also has an optimization for small `IsoSubspace` by making them backed by `PreciseAllocation`[9]). This is a security hardening feature that makes use-after-free-based attacks harder[10].

`IsoSubspace` is mostly a simplified `CompleteSubspace`, so we will ignore it for the purpose of this post. `CompleteSubspace` is the one that handles the common case: small allocations, and `PreciseAllocation` is mostly the rare slow path for large allocations.

Generational GC Basics

In JSC's generational GC model, the heap consists of a small "new space" (eden), holding the newly allocated objects, and a large "old space" holding the older objects that have survived one GC cycle. Each GC cycle is either an *eden GC* or a *full GC*. New objects are allocated in the eden. When the eden is full, an eden GC is invoked to garbage-collect the unreachable objects in eden. All the surviving objects in eden are then considered to be in the old space[11]. To reclaim objects in the old space, a full GC is needed.

The effectiveness of the above scheme relies on the so-called "generational hypothesis":

1. Most objects collected by the GC are young objects (died when they are still in eden), so an eden GC (which only collects the eden) is sufficient to reclaim most newly allocated memory.

2. Pointers from old space to eden is much rarer than pointers from eden to old space or pointers from eden to eden, so an eden GC's runtime is approximately linear to the size of the eden, as it only needs to start from a small subset of the old space. This implies that the cost of GC can be amortized by the cost of allocation.

Inlined vs. Outlined Metadata: Why?

Practically every GC scheme uses some kind of metadata to track which objects are alive. In this section, we will explain how the GC metadata is stored in JSC, and the motivation behind its design.

In JSC, every object managed by the GC carries the following metadata:

1. Every object managed by the GC inherits the JSCell class, which contains a 1-byte member `cellState`. This `cellState` is a color marker with two colors: white and black[12].

2. Every object also has two out-of-object metadata bits: `isNew`[13] and `isMarked`. For objects allocated by `PreciseAllocation`, the bits reside in the GC header. For objects allocated by `CompleteSubspace`, the bits reside in the block footer.

This may seem odd at first glance since `isNew` and `isMarked` could have been stored in the unused bits of `cellState`. However, this is intentional.

The inlined metadata `cellState` is easy to access for the mutator thread (the thread executing JavaScript code), since it is just a field in the object. However, it has bad memory locality for the GC and allocators, which need to quickly traverse through all the metadata of all objects in some block owned by `CompleteSubspace` (which is the common case). Outlined metadata have the opposite performance characteristics: they are more expensive to access for the mutator thread, but since they are aggregated into bitvectors and stored in the block footer of each block, GC and allocators can traverse them really fast.

So JSC keeps both inlined and outlined metadata to get the better of both worlds: the mutator thread's fast path will only concern the inlined `cellState`, while the GC and allocator logic can also take advantage of the memory locality of the outlined bits `isNew` and `isMarked`.

Of course, the cost of this is a more complex design… so we have to unfold it bit by bit.

A Really Naive Stop-the-World Generational GC

Let's start with a really naive design just to understand what is needed. We will design a generational, but stop-the-world (i.e. not incremental nor concurrent) GC, with no performance optimizations at all. In this design, the mutator side transfers control to the GC subsystem at a "safe point"[14] to start a GC cycle (eden or full). The GC subsystem performs the GC cycle from the beginning to the end (as a result, the application cannot run during this potentially long period, thus "stop-the-world"), and then transfer control back to the mutator side.

For this purpose, let's temporarily forget about `CompleteSubspace`: it is an optimized version of `PrecisionAllocation` for small allocations, and while it is an important optimization, it's easier to understand the GC algorithm without it.

It turns out that in this design, all we need is one `isMarked` bit. The `isMarked` bit will indicate if the object is reachable at the end of the last GC cycle (and consequently, is in the old space, since any object that survived a GC cycle is in old space). All objects are born with `isMarked = false`.

The GC will use a breadth-first search to scan and mark objects. For full GC, we want to reset all `isMarked` bits to `false` at the beginning, and do a BFS to scan and mark all objects reachable from GC roots. Then all the unmarked objects are known to be dead. For an eden GC, we only want to scan the eden space. Fortunately, all objects in the old space are already marked at the end of the previous GC cycle, so they are naturally ignored by the BFS, so we can simply reuse the same BFS algorithm in full GC. In pseudo-code:

Eden GC preparation phase: no work is needed.

Full GC preparation phase[15]:

```
for (JSCell* obj : heap)
    obj->isMarked = false;
```

Eden/Full GC marking phase:

```
while (!queue.empty()) {
    JSCell* obj = queue.pop();
    obj->ForEachChild([&](JSCell* child) {
        if (!child->isMarked) {
            child->isMarked = true;
            queue.push(child);
        }
    });
}
```

Eden/Full GC collection phase:

```
// One can easily imagine an optimization to make eden collection
// traverse only the eden space. We ignore it for simplicity.
for (JSCell* obj : heap) {
    if (!obj->isMarked)
        free(obj);
}
```

But where does the scan start, so that we can scan through every reachable object? For full GC, the answer is clear: we just start the scan from all GC roots[16]. However, for an eden GC, in order to reliably scan through all reachable objects, the situation is slightly more complex:

1. Of course, we still need to push the GC roots to the initial queue.
2. If an object in the old space contains a pointer to an object in eden, we need to put the old space object to the initial queue[17].

The invariant for the second case is maintained on the mutator side. Specifically, whenever one writes a pointer slot of some object `A` in the heap to point to another object `B`, one needs to check if `A.isMarked` is `true` and `B.isMarked` is `false`. If so, one needs to put `A` into a "remembered set". An eden GC must treat the objects in the remembered set as if they were GC roots. This is called a `WriteBarrier`. In pseudo-code:

```
// Executed after writing a pointer to 'dst' into a field of 'obj'
if (obj->isMarked && !dst->isMarked)
    addToRememberedSet(obj);
```

Getting Incremental

A stop-the-world GC isn't optimal for production use. A GC cycle (especially a full GC cycle) can take a long time. Since the mutator (application logic) cannot run during the stop-the-world period, the application would appear irresponsive to the user, which can be a very bad user experience for long pauses.

A natural way to shorten this irresponsive period is to run GC incrementally: at safe points, the mutator transfers control to the GC. The GC only runs for a short time, doing a portion of the work for the current GC cycle (eden or full), then return control to the mutator. This way, each GC cycle is split into many small steps, so the irresponsive periods are less noticeable to the user.

Incremental GC poses a few new challenges to the GC scheme.

The first challenge is the extra interference between the GC and the mutator: the mutator, namely the allocator and the `WriteBarrier`, must be prepared to see states arisen from a partially-completed GC cycle. And the GC side must correctly mark all reachable objects despite changes made by the mutator side in between.

Specifically, our full GC must change: imagine that the full GC scanned some object `o` and handed back control to mutator, then the mutator changed a field of `o` to point to some other object `dst`. The object `dst` must not be missed from scanning. Fortunately, in such a case `o` will be `isMarked` and `dst` will be `!isMarked` (if `dst` has `isMarked` then it has been scanned, so there's nothing to worry about), so `o` will be put into the remembered set.

Therefore, for a full GC to function correctly in the incremental GC scheme, it must consider the remembered set as a GC root as well, just like the eden GC.

The other parts of the algorithm as of now can remain unchanged (we leave the proof of correctness as an exercise for the reader). Nevertheless, "what happens if a GC cycle is run partially?" is something that we must keep in mind as we add more optimizations.

The second challenge is that the mutator side can repeatedly put an old space object into the remembered set, and result in redundant work for the GC: for example, the GC popped some object `o` in the remembered set, traversed from it, and handed over control to mutator. The mutator modified `o` again, putting it back to the remembered set. If this happens too often, the incremental GC could do a lot more work than a stop-the-world GC.

The situation will get even worse once we make our GC concurrent: in a concurrent GC, since the GC is no longer stealing CPU time from the mutator, the mutator gets higher throughput, thus will add even more objects into the remembered set. In fact, JSC observed up to 5x higher memory consumption without any mitigation. Therefore, two techniques are employed to mitigate this issue.

The first and obvious mitigation is to have the GC scan the remembered set last: only when the queue has otherwise been empty do we start popping from the remembered set. The second mitigation employed by JSC is a technique called *Space-Time Scheduler*. In short, if it observes that the mutator was allocating too fast, the mutator would get decreasingly less time quota to run so the GC can catch up (and in the extreme case, the mutator would get zero time quota to run, so it falls back to the stop-the-world approach). Filip Pizlo's blog post has explained it very clearly, so feel free to take a look if you are interested.

Anyways, let's update the pseudo-code for the eden/full GC marking phase:

```
while (!queue.empty() || !rmbSet.empty()) {
    // Both eden GC and full GC needs to consider the remembered set
    // Prioritize popping from queue, pop remembered set last
    JSCell* obj = !queue.empty() ? queue.pop() : rmbSet.pop();
    obj->ForEachChild([&](JSCell* child) {
        if (!child->isMarked) {
            child->isMarked = true;
            queue.push(child);
        }
    });
}
```

Incorporate in CompleteSubspace

It's time to get our `CompleteSubspace` allocator back so we don't have to suffer the huge per-object GC header overhead incurred by `PreciseAllocation`.

For `PreciseAllocation`, the actual memory management work is done by `malloc`: when the mutator wants to allocate an object, it just `malloc`s it, and when the GC discovers a dead object, it just `free`s it.

`CompleteSubspace` introduces another complexity, as it only allocates/deallocates memory from malloc at 16KB-block level, and does memory management itself to divide the blocks into cells that it serves to the application. Therefore, it has to track whether each of its cells is available for allocation. The mutator allocates from the available cells, and the GC marks dead cells as available for allocation again.

The `isMarked` bit is not enough for the `CompleteSubspace` allocator to determine if a cell contains a live object or not: newly allocated objects have `isMarked = false` but are clearly live objects. Therefore, we need another bit.

In fact, there are other good reasons that we need to support checking if a cell contains a live object or not. A canonical example is the conservative stack scanning: JSC does not precisely understand the layout of the stack, so it needs to treat everything on the stack that could be pointers and pointing to live objects as a GC root, and this involves checking if a heap pointer points to a live object or not.

One can easily imagine some kind of `isLive` bit that is `true` for all live objects, which is only flipped to `false` by the GC when the object is dead. However, JSC employed a slightly different scheme, which is needed to facilitate optimizations that we will mention later.

As you have seen earlier, the bit used by JSC is called `isNew`.

However, keep in mind: you should **not** think of `isNew` as a bit that tells you **anything** related to its name, or indicates anything by itself. You should think of it as a helper bit, which sole purpose is that, when working together with `isMarked`, they tell you if a cell contains a live object or not. This thinking mode will be more important in the next section when we introduce logical versioning.

The core invariant around `isNew` and `isMarked` is:

1. At **any** moment, an object is dead iff its `isNew = false` and `isMarked = false`.

If we were a stop-the-world GC, then to maintain this invariant, we only need the following:

1. When an object is born, it has `isNew = true` and `isMarked = false`.

2. At the end of each eden or full GC cycle, we set `isNew` of all objects to `false`.

Then, all newly-allocated objects are live because its `isNew` is `true`. At the end of each GC cycle, an object is live iff its `isMarked` is `true`, so after we set `isNew` to `false` (due to rule 2), the invariant on what is a dead object is maintained, as desired.

However, in an incremental GC, since the state of a partially-run GC cycle can be exposed to mutator, we need to ensure that the invariant holds in this case as well.

Specifically, in a full GC, we reset all `isMarked` to `false` at the beginning. Then, during a partially-run GC cycle, the mutator may see a live object with both `isMarked = false` (because it has not been marked by the GC yet), and `isNew = false` (because it has survived one prior GC cycle). This violates our invariant.

To fix this, at the beginning of a full GC, we additionally do `isNew |= isMarked` before clearing `isMarked`. Now, during the whole full GC cycle, all live objects must have `isNew = true`[18], so our invariant is maintained. At the end of the cycle, all `isNew` bits are cleared, and as a result, all the

unmarked objects become dead, so our invariant is still maintained as desired. So let's update our pseudo-code:

Eden GC preparation phase: no work is needed.

Full GC preparation phase:

```
// Do 'isNew |= isMarked, isMarked = false' for all
// PreciseAllocation and all cells in CompleteSubspace

for (PreciseAllocation* pa : allPreciseAllocations) {
    pa->isNew |= pa->isMarked;
    pa->isMarked = false;
}

for (BlockFooter* block : allCompleteSubspaceBlocks) {
    for (size_t cellId = 0; cellId < block->numCells; cellId++) {
        block->isNew[cellId] |= block->isMarked[cellId];
        block->isMarked[cellId] = false;
    }
}
```

Eden/Full GC collection phase:

```
// Update 'isNew = false' for CompleteSubspace cells
for (BlockFooter* block : allCompleteSubspaceBlocks) {
    for (size_t cellId = 0; cellId < block->numCells; cellId++) {
        block->isNew[cellId] = false;
    }
}

// For PreciseAllocation, in addition to updating 'isNew = false',
// we also need to free the dead objects
for (PreciseAllocation* pa : allPreciseAllocations) {
    pa->isNew = false;
    if (!pa->isMarked)
        free(pa);
}
```

In `CompleteSubspace` allocator, to check if a cell in a block contains a live object (if not, then the cell is available for allocation):

```
bool cellContainsLiveObject(BlockFooter* block, size_t cellId) {
    return block->isMarked[cellId] || block->isNew[cellId];
}
```

Logical Versioning: Do Not Sweep!

We are doing a lot of work at the beginning of a full GC cycle and at the end of any GC cycle, since we have to iterate through all the blocks in `CompleteSubspace` and update their `isMarked` and `isNew` bits. Despite that the bits in one block are clustered into bitvectors thus have good memory locality, this could still be an expensive operation, especially after we have a concurrent GC (as this stage cannot be made concurrent). So we want something better.

The optimization JSC employs is logical versioning. Instead of physically clearing all bits in all blocks for every GC cycle, we only bump a global "logical version", indicating that all the bits are logically cleared (or updated). Only when we actually need to mark a cell in a block during the marking phase do we then physically clear (or update) the bitvectors in this block.

You may ask: why bother with logical versioning, if in the future we still have to update the bitvectors physically anyway? There are two good reasons:

1. If all cells in a block are dead (either died out during this GC cycle[19], or already dead before this GC cycle), then we will never mark anything in the block, so logical versioning enabled us to avoid the work altogether. This also implies that at the end of each GC cycle, it's unnecessary to figure out which blocks become completely empty, as logical versioning makes sure that these empty blocks will not cause overhead to future GC cycles.

2. The marking phase can be done concurrently with multiple threads *and* while the mutator thread is running (our scheme isn't concurrent now, but we will do it soon), while the preparation / collection phase must be performed with the mutator stopped. Therefore, shifting the work to the marking phase reduces GC latency in a concurrent setting.

There are two global version number `g_markVersion` and `g_newVersion`[20]. Each block footer also stores its local version number `l_markVersion` and `l_newVersion`.

Let's start with the easier case: the logical versioning for the `isNew` bit.

If you revisit the pseudo-code above, in GC there is only one place where we write `isNew`: at the end of each GC cycle, we set all the `isNew` bits to `false`. Therefore, we simply bump `g_newVersion` there instead. A local version `l_newVersion` smaller than `g_newVersion` means that all the `isNew` bits in this block have been logically cleared to `false`.

When the `CompleteSubspace` allocator allocates a new object, it needs to start with `isNew = true`. One can clearly do this directly, but JSC did it in a trickier way that involves a block-level bit named `allocated` for slightly better performance. This is not too interesting, so I deferred it to the end of the post, and our scheme described here right now will not employ this optimization (but is otherwise intentionally kept semantically equivalent to JSC):

1. When a `BlockDirectory` starts allocating from a new block, it update the the block's `l_newVersion` to `g_newVersion`, and set `isNew` to `true` for all already-allocated cells (as the block may not be fully empty), and `false` for all free cells.

2. Whenever it allocates a cell, it sets its `isNew` to true.

Why do we want to bother setting `isNew` to `true` for all already-allocated cells in the block? This is to provide a good property. Since we bump `g_newVersion` at the end of every GC cycle, due to the scheme above, for any block with latest `l_newVersion`, a cell is live if and only if its `isNew` bit is set. Now, when checking if a cell is live, if its `l_newVersion` is the latest, then we can just return `isNew` without looking at `isMarked`, so our logic is simpler.

The logical versioning for the `isMarked` bit is similar. At the beginning of a full GC cycle, we bump the `g_markVersion` to indicate that all mark bits are logically cleared. Note that the global version is not bumped for eden GC, since eden GC does not clear `isMark` bits.

There is one extra complexity: the above scheme would break down in an incremental GC. Specifically, *during* a full GC cycle, we have logically cleared the `isMarked` bit, but we also didn't do anything to the `isNew` bit, so all cells in the old space would appear dead to the allocator. In our old scheme without logical versioning, this case is prevented by doing `isNew |= isMarked` at the start of the full GC, but we cannot do it now with logical versioning.

JSC solves this problem with the following clever trick: *during* a full GC, we should also accept `l_markVersion` that is off-by-one. In that case, we know the `isMarked` bit accurately reflects whether or not a cell is live, since that is the result of the last GC cycle. If you are a bit confused, take a look at

footnote[21] for a more elaborated case discussion. It might also help to take a look at the comments in the pseudo-code below:

```
bool cellContainsLiveObject(BlockFooter* block, size_t cellId) {
    if (block->l_newVersion == g_newVersion) {
        // A latest l_newVersion indicates that the cell is live if
        // and only if its 'isNew' bit is set, so we don't need to
        // look at the 'isMarked' bit even if 'isNew' is false
        return block->isNew[cellId];
    }

    // Now we know isNew bit is logically false, so we should
    // look at the isMarked bit to determine if the object is live
    if (isMarkBitLogicallyCleared(block)) {
        // The isMarked bit is logically false
        return false;
    }

    // The isMarked bit is valid and accurately tells us if
    // the object is live or not
    return block->isMarked[cellId];
}


// Return true if the isMarked bitvector is logically cleared
bool isMarkBitLogicallyCleared(BlockFooter* block) {
    if (block->l_markVersion == g_markVersion) {
        // The mark version is up-to-date, so not cleared
        return false;
    }

    if (IsFullGcRunning() && IsGcInMarkingPhase() &&
        block->l_markVersion == g_markVersion - 1) {
        // We are halfway inside a full GC cycle's marking phase,
        // and the mark version is off-by-one, so the old isMarked bit
        // should be accepted, and it accurately tells us if the
        // object is live or not
        return false;
    }

    return true;
}
```

Before we mark an object in `CompleteSubspace`, we need to update the `l_markVersion` of the block holding the cell to the latest, and materialize the `isMarked` bits of all cells in the block. That is, we need to run the logic at the full GC preparation phase in our old scheme: `isNew |= isMarked`, `isMarked = false` for all cells in the block. This is shown below.

```
// Used by GC marking phase to mark an object in CompleteSubspace
void markObject(BlockFooter* block, size_t cellId) {
    aboutToMark(block);
    block->isMarked[cellId] = true;
}


// Materialize 'isMarked' bits if needed
// To do this, we need to execute the operation at full GC
// prepare phase: isNew |= isMarked, isMarked = false
void aboutToMark(BlockFooter* block) {
    if (block->l_markVersion == g_markVersion) {
        // Our mark version is already up-to-date,
        // which means it has been materialized before
        return;
    }
```

```
    // Check if the isMarked bit is logically cleared to false.
    // The function is defined in the previous snippet.
    if (isMarkBitLogicallyCleared(block)) {
        // This means that the isMarked bitvector should
        // be treated as all false. So operation isNew |= isMarked
        // is no-op, so all we need to do is isMarked = false
        for (size_t cellId = 0; cellId < block->numCells; cellId++) {
            block->isMarked[cellId] = false;
        }
    } else {
        // The 'isMarked' bit is not logically cleared. Now let's
        // check if the 'isNew' bit is logically cleared.
        if (block->l_newVersion < g_newVersion) {
            // The isNew bitvector is logically cleared and should be
            // treated as false. So operation isNew |= isMarked becomes
            // isNew = isMarked (note that executing |= is incorrect
            // beacuse isNew could physically contain true!)
            for (size_t cellId = 0; cellId < block->numCells; cellId++) {
                block->isNew[cellId] = block->isMarked[cellId];
                block->isMarked[cellId] = false;
            }

            // We materialized isNew, so update it to latest version
            block->l_newVersion = g_newVersion;
        } else {
            // The l_newVersion is latest, which means that the cell is
            // live if and only if its isNew bit is set.
            // Since isNew already reflects liveness, we do not have to
            // perform the operation isNew |= isMarked (and in fact, it
            // must be a no-op since no dead cell can have isMarked =
            // true). So we only need to do isMarked = false
            for (size_t cellId = 0; cellId < block->numCells; cellId++) {
                block->isMarked[cellId] = false;
            }
        }
    }

    // We finished materializing isMarked, so update the version
    block->l_markVersion = g_markVersion;
}
```

A fun fact: despite that what we conceptually want to do above is `isNew |= isMarked`, the above code never performs a `|=` at all 🙄

And also, let's update the pseudo-code for the preparation GC logic:

Eden GC preparation phase: no work is needed.

Full GC preparation phase:

```
// For PreciseAllocation, we still need to manually do
// 'isNew |= isMarked, isMarked = false' for every allocation
for (PreciseAllocation* pa : allPreciseAllocations) {
    pa->isNew |= pa->isMarked;
    pa->isMarked = false;
}

// For CompleteSubspace, all we need to do is bump the
// global version for the 'isMarked' bit
g_markVersion++;
```

Eden/Full GC collection phase:

```
// For PreciseAllocation, we still need to manually
// update 'isNew = false' for each allocation, and also
// free the object if it is dead
for (PreciseAllocation* pa : allPreciseAllocations) {
    pa->isNew = false;
    if (!pa->isMarked)
        free(pa);
}

// For CompleteSubspace, all we need to do is bump the
// global version for the 'isNew' bit
g_newVersion++;
```

With logical versioning, the GC no longer sweeps the `CompleteSubspace` blocks to reclaim dead objects: the reclamation happens lazily, when the allocator starts to allocate from the block. This, however, introduces an unwanted side-effect. Some objects use manual memory management internally: they own additional memory that are not managed by the GC, and have C++ destructors to free that memory when the object is dead. This improves performance as it reduces the work of the GC. However, now we may not immediately sweep dead objects and run destructors, so the memory that is supposed to be freed by the destructor could be kept around indefinitely if the block is never allocated from. To mitigate this issue, JSC will also periodically sweep blocks and run the destructors of the dead objects. This is implemented by `IncrementalSweeper`, but we will not go into details.

To conclude, logical versioning provides two important optimizations to the GC scheme:

1. The so-called "sweep" phase of the GC (to find and reclaim dead objects) is removed for `CompleteSubspace` objects. The reclamation is done lazily. This is clearly better than sweeping through the block again and again in every GC cycle.
2. The full GC does not need to reset all `isMarked` bits in the preparation phase, but only lazily reset them in the marking phase by `aboutToMark`: this not only reduces work, but also allows the work to be done in parallel and concurrently while the mutator is running, after we make our GC scheme concurrent.

Optimizing WriteBarrier: The cellState Bit

As we have explained earlier, whenever the mutator modified a pointer of a marked object `o` to point to an unmarked object, it needs to add `o` to the "remembered set", and this is called the `WriteBarrier`. In this section, we will dig a bit deeper into the `WriteBarrier` and explain the optimizations around it.

The first problem with our current `WriteBarrier` is that the `isMarked` bit resides in the block footer, so retrieving its value requires quite a few computations from the object pointer. Also it doesn't sit in the same CPU cache line as the object, which makes the access even slower. This is undesirable as the cost is paid for every `WriteBarrier`, regardless of if we add the object to the remembered set.

The second problem is our `WriteBarrier` will repeatedly add the same object `o` to the remembered set every time it is run. The obvious solution is to make `rememberedSet` a hash set to de-duplicate the objects it contains, but doing a hash lookup to check if the object already exists is far too expensive.

This is where the last metadata bit that we haven't explained yet: the `cellState` bit comes in, which solves both problems.

Instead of making `rememberedSet` a hash table, we reserve a byte (though we only use 1 bit of it) named `cellState` in every object's object header, to indicate if we might need to put the object into the remembered set in a `WriteBarrier`. Since this bit resides in the object header as an object field (instead of in the block footer), it's trivially accessible to the mutator who has the object pointer.

`cellState` has two possible values: `black` and `white`. The most important two invariants around `cellState` are the following:

1. For any object with `cellState = white`, it is guaranteed that the object does not need to be added to remembered set.
2. Unless *during* a full GC cycle, all `black` (live) objects have `isMarked = true`.

Invariant 1 serves as a fast-path: `WriteBarrier` can return immediately if our object is `white`, and checking it only requires one load instruction (to load `cellState`) and one comparison instruction to validate it is `white`.

However, if the object is `black`, a slow-path is needed to check whether it is actually needed to add the object to the remembered set.

Let's look at our new `WriteBarrier`:

```
// Executed after writing a pointer to 'dst' into a field of 'obj'
void WriteBarrier(JSCell* obj) {
    if (obj->cellState == black)
        WriteBarrierSlowPath(obj);
}
```

The first thing to notice is that the `WriteBarrier` is no longer checking if `dst` (the object that the pointer points to) is marked or not. Clearly this does not affect the correctness: we are just making the criteria less restrictive. However, the performance impact of removing this `dst` check is a tricky question without a definite answer, even for JSC developers. Through some preliminary testing, their conclusion is that adding back the `!isMarked(dst)` check slightly regresses performance. They have two hypotheses. First, by not checking `dst`, more objects are put into the remembered set and need to be scanned by the GC, so the total amount of work increased. However, the mutator's work probably decreased, as it does fewer checks and touches fewer cache lines (by not touching the outlined `isMarked` bit). Of course such benefit is offset because the mutator is adding more objects into the remembered set, but this isn't too expensive either, as the remembered set is only a segmented vector. The GC has to do more work, as it needs to scan and mark more objects. However, after we make our scheme concurrent, the marking phase of the GC can be done concurrently as the mutator is running, so the latency is probably[22] hidden. Second, JSC's DFG compiler has an optimization pass that coalesces barriers on the same object together, and it is harder for such barriers to check all the `dsts`.

The interesting part is how the invariants above are maintained by the relavent parties. As always, there are three actors: the mutator (`WriteBarrier`), the allocator, and the GC.

The interaction with the allocator is the simplest. All objects are born `white`. This is correct because newly-born objects are not marked, so have no reason to be remembered.

The interaction with GC is during the GC marking phase:

1. When we mark an object and push it into the queue, we set its `cellState` to `white`.
2. When we pop an object from the queue, before we start to scan its children, we set its `cellState` to `black`.

In pseudo-code, the Eden/Full GC marking phase now looks like the following (Line 5 and Line 9 are the newly-added logic to handle `cellState`, other lines unchanged):

```
while (!queue.empty() || !rmbSet.empty()) {
    // Both eden GC and full GC needs to consider remembered set
    // Prioritize popping from queue, pop remembered set last
    JSCell* obj = !queue.empty() ? queue.pop() : rmbSet.pop();
    obj->cellState = black;           // <----------------- newly added
```

```
    obj->ForEachChild([&](JSCell* child) {
        if (!child->isMarked) {
            markObject(child);
            child->cellState = white; // <----------------- newly added
            queue.push(child);
        }
    });
}
```

Let's argue why the invariant is maintained by the above code.

1. For invariant 1, note that in the above code, an object is `white` only if it is inside the queue (as once it's popped out, it becomes `black` again), pending scanning of its children. Therefore, it is guaranteed that the object will still be scanned by the GC later, so we don't need to add the object to remembered set, as desired.
2. For invariant 2, at the end of any GC cycle, any live object is marked, which means it has been scanned, so it is `black`, as desired.

Now let's look at what `WriteBarrierSlowPath` should do. Clearly, it's correct if it simply unconditionally add the object to remembered set, but that also defeats most of the purpose of `cellState` as an optimization mechanism: we want something better. A key use case of `cellState` is to prevent adding an object into the remembered set if it is already there. Therefore, after we put the object into the remembered set, we will set its `cellState` to `white`, like shown below.

```
void WriteBarrierSlowPath(JSCell* obj) {
    obj->cellState = white;
    addToRememberedSet(obj);
}
```

Let's prove why the above code works. Once we added an object to remembered set, we set it to `white`. We don't need to add the same object into the remembered set until it gets popped out from the set by GC. But when GC pops out the object, it would set its `cellState` back to `black`, so we are good.

JSC employed one more optimization. During a full GC, we might see a `black` object that has `isMarked = false` (note that this is the only possible case that the object is unmarked, due to invariant 2). In this case, it's unnecessary to add the object to remembered set, since the object will eventually be scanned in the future (or it became dead some time later before it was scanned, in which case we are good as well). Furthermore, we can flip it back to `white`, so we don't have to go into this slow path the next time a `WriteBarrier` on this object runs. To sum up, the optimized version is as below:

```
void WriteBarrierSlowPath(JSCell* obj) {
    if (IsFullGcRunning()) {
        if (!isMarked(obj)) {
            // Do not add the object to remembered set
            // In addition, set cellState to white so this
            // slow path is not triggered on the next run
            obj->cellState = white;
            return;
        }
    } else {
        assert(isMarked(obj)); // due to invariant 2
    }

    obj->cellState = white;
    addToRememberedSet(obj);
}
```

Getting Concurrent and Getting Wild

At this point, we already have a very good incremental and generational garbage collector: the mutator, allocator and GC all have their respective fast-paths for the common cases, and with logical versioning, we avoided redundant work as much as possible. In my humble opinion, this is a good balance point between performance and engineering complexity.

However, because JSC is one of the core drivers of performance in Safari, it's unsurprising that performance is a top priority, even at the cost of engineering complexity. To squeeze out every bit of performance, JSC made their GC concurrent. This is no easy feat: due to the nature of GCs, it's often too slow to use locks to protect against certain race conditions, so extensive lock-free programming is employed.

But once lock-free programming is involved, one starts to get into all sorts of architecture-dependent memory reordering problems. x86-64 is the more strict architecture: it only requires `StoreLoadFence()`, and it provides TSO-like semantics. JSC also supports ARM64 CPUs, which has even fewer guarantees: load-load, load-store, store-load, and store-store can all be reordered by the CPU, so a lot more operations need fences. As if things were not bad enough, for performance reasons, JSC often avoids using memory fences on ARM64. They have the so-called `Dependency` class, which creates an implicit CPU data dependency on ARM64 through some scary assembly hacks, so they can get the desired memory ordering for a specific data-flow without paying the cost of a memory fence. As you can imagine, with all of these complications and optimizations, the code can become difficult to read.

So due to my limited expertise, it's unsurprising if I missed to explain or mis-explained some important race conditions in the code, especially some ARM64-specific ones: if you spotted any issue in this post, please let me know.

Let's go through the concurrency assumptions first. JavaScript is a single-threaded language, so there is always only one mutator thread[23]. Apart from the mutator thread, JSC has a bunch of compilation threads, a GC thread, and a bunch of marking threads. Only the GC marking phase is concurrent: during which the mutator thread, the compiler threads, and a bunch of marking threads are concurrently running (yes, the marking itself is also done in parallel). However, all the other GC phases are run with the mutator thread and compilation threads stopped.

Some Less Interesting Issues

First of all, clearly the `isMarked` and `isNew` bitvector must be made safe for concurrent access, since multiple threads (including marking threads and mutator) may concurrently update it. Using CAS with appropriate retry/bail mechanism is enough for the bitvector itself.

`BlockFooter` is harder, and needs to be protected with a lock: multiple threads could be simultaneously calling `aboutToMark()`, so `aboutToMark()` must be guarded. For the reader side (the `isMarked()` function, which involves first checking if `l_markVersion` is latest, then reading the `isMarked` bitvector), in x86-64 thanks to x86-TSO, one does not need a lock or any memory fence (as long as `aboutToMark` takes care to update `l_markVersion` after the bitvector). In ARM64, since load-load reordering is allowed, a `Dependency` is required.

Making the `cellContainsLiveObject` (or in JSC jargon, `isLive`) check lock-free is harder, since it involves potentially reading both the `isMarked` bit and the `isNew` bit. JSC employs optimistic locking to provide a fast-path. This is not very different from an optimistic locking scheme you can find in a textbook, so I won't dive into the details.

Of course, there are a lot more subtle issues to change. Almost all the pseudo-code above needs to be adapted for concurrency, either by using a lock or CAS, or by using some sort of memory barrier and

concurrency protocol to ensure that the code works correctly under races. But now let's turn to some more important and tricky issues.

The Race Between WriteBarrier and Marking

One of the most important races is the race between `WriteBarrier` and GC's marking threads. The marking threads and the mutator thread can access the `cellState` of an object concurrently. For performance reasons, a lock is infeasible, so a race condition arises.

It's important to note that we call `WriteBarrier` **after** we have written the pointer into the object. This is not only more convenient to use (especially for JIT-generated code), but also allows a few optimizations: for example, in certain cases, multiple writes to the same object may only call `WriteBarrier` once at the end.

With this in mind, let's analyze why our current implementation is buggy. Suppose `o` is an object, and the mutator wants to store a pointer to another object `target` into a field `f` of `o`. The marking logic of the GC wants to scan `o` and append its children into the queue. We need to make sure that GC will observe the `o->target` pointer link.

Let's first look at the correct logic:

Mutator (WriteBarrier)	**GC (Marker)**
`Store(o.f, target)`	`Store(o.cellState, black)`
`StoreLoadFence() // WriteBarrier begin`	`StoreLoadFence()`
`t1 = Load(o.cellState)`	`t2 = Load(o.f) // Load a children of o`
`if (t1 == black): WriteBarrierSlowPath(o)`	`Do some check to t2 and push it to queue`

This is mostly just a copy of the pseudocode in the above sections, except that we have two `StoreLoadFence()`. A `StoreLoadFence()` guarantees that no `LOAD` after the fence may be executed by the CPU out-of-order engine until all `STORE` before the fence have completed. Let's first analyze what could go wrong without either of the fences.

Just to make things perfectly clear, the precondition is `o.cellState = white` (because `o` is in the GC's queue) and `o.f = someOldValue`.

What could go wrong if the mutator `WriteBarrier` doesn't have the fence? Without the fence, the CPU can execute the `LOAD` in line 3 before the `STORE` in line 1. Then, in the following interleaving:

1. [Mutator Line 3] t1 = Load(o.cellState) // t1 = white
2. [GC Line 1] Store(o.cellState, black)
3. [GC Line 3] t2 = Load(o.f) // t2 = some old value
4. [Mutator Line 1] Store(o.f, target)

Now, the mutator did not add `o` to remembered set (because `t1` is `white`, not `black`), and `t2` in GC is the old value in `o.f` instead of `target`, so GC did not push `target` into the queue. So the pointer link from `o` to `target` is missed in GC. This can result in `target` being wrongly reclaimed despite it is live.

And what could go wrong if the GC marking logic doesn't have the fence? Similarly, without the fence, the CPU can execute the `LOAD` in line 3 before the `STORE` in line 1. Then, in the following interleaving:

1. [GC Line 3] t2 = Load(o.f) // t2 = some old value
2. [Mutator Line 1] Store(o.f, target)
3. [Mutator Line 3] t1 = Load(o.cellState) // t1 = white
4. [GC Line 1] Store(o.cellState, black)

Similar to above, mutator sees `t1 = white` and GC sees `t2 = oldValue`. So `o` is not added to remembered set, and `target` is not pushed into the queue, the pointer link is missed.

Finally, let's analyze why the code behaves correctly if both fences are present. Unfortunately there is not a better way than manually enumerating all the interleavings. Thanks to the fences, `Mutator Line 1` must execute before `Mutator Line 3`, and `GC Line 1` must execute before `GC Line 3`, but the four lines can otherwise be reordered arbitrarily. So there are `4! / 2! / 2! = 6` possible interleavings. So let's go!

Interleaving 1:

1. [Mutator Line 1] Store(o.f, target)
2. [Mutator Line 3] t1 = Load(o.cellState) // t1 = white
3. [GC Line 1] Store(o.cellState, black)
4. [GC Line 3] t2 = Load(o.f) // t2 = target

In this interleaving, the mutator did not add `o` to remembered set, but the GC sees `target`, so it's fine.

Interleaving 2:

1. [GC Line 1] Store(o.cellState, black)
2. [GC Line 3] t2 = Load(o.f) // t2 = some old value
3. [Mutator Line 1] Store(o.f, target)
4. [Mutator Line 3] t1 = Load(o.cellState) // t1 = black

In this interleaving, the GC saw the old value, but the mutator added `o` to the remembered set, so the GC will eventually drain from the remembered set and scan `o` again, at which time it will see the correct new value `target`, so it's fine.

Interleaving 3:

1. [Mutator Line 1] Store(o.f, target)
2. [GC Line 1] Store(o.cellState, black)
3. [Mutator Line 3] t1 = Load(o.cellState) // t1 = black
4. [GC Line 3] t2 = Load(o.f) // t2 = target

In this interleaving, the GC saw the new value `target`, nevertheless, the mutator saw `t1 = black` and added `o` to the remembered set. This is unfortunate since the GC will scan `o` again, but it doesn't affect correctness.

Interleaving 4:

1. [Mutator Line 1] Store(o.f, target)
2. [GC Line 1] Store(o.cellState, black)
3. [GC Line 3] t2 = Load(o.f) // t2 = target
4. [Mutator Line 3] t1 = Load(o.cellState) // t1 = black

Same as Interleaving 3.

Interleaving 5:

1. [GC Line 1] Store(o.cellState, black)
2. [Mutator Line 1] store(o.f, target)
3. [Mutator Line 3] t1 = Load(o.cellState) // t1 = black
4. [GC Line 3] t2 = Load(o.f) // t2 = target

Same as Interleaving 3.

Interleaving 6:

1. [GC Line 1] Store(o.cellState, black)
2. [Mutator Line 1] Store(o.f, target)
3. [GC Line 3] t2 = Load(o.f) // t2 = target
4. [Mutator Line 3] t1 = Load(o.cellState) // t1 = black

Same as Interleaving 3.

This proves that with the two `StoreLoadFence()`, our code is no longer vulnerable to the above race condition.

Another Race Condition Between WriteBarrier and Marking

The above fix alone is not enough: there is another race between `WriteBarrier` and GC marking threads. Recall that in `WriteBarrierSlowPath`, we attempt to flip the object back to `white` if we saw it is not marked (this may happen during a full GC), as illustrated below:

```
... omitted ...
if (!isMarked(obj)) {
    obj->cellState = white;
    return;
}
... omitted ...
```

It turns out that, after setting the object `white`, we need to do a `StoreLoadFence()`, and check again if the object is marked. If it becomes marked, we need to set `obj->cellState` back to `black`.

Without the fix, the code is vulnerable to the following race:

1. [Precondition] `o.cellState = black` and `o.isMarked = false`
2. [WriteBarrier] Check isMarked() // see false
3. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
4. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
5. [WriteBarrier] Store(o.cellState, white)
6. [Postcondition] `o.cellState = white` and `o.isMarked = true`

The post-condition is bad because `o` will not be added to the remembered set in the future, despite that it needs to be (as the GC has already scanned it).

Let’s now prove why the code is correct when the fix is applied. Now the `WriteBarrier` logic looks like this:

1. [WriteBarrier] Store(o.cellState, white)
2. [WriteBarrier] t1 = isMarked()

3. [WriteBarrier] if (t1 == true): Store(o.cellState, black)

Note that we omitted the first "Check isMarked()" line because it must be the first thing executed in the interleaving, as otherwise the `if`-check won't pass at all.

The three lines in `WriteBarrier` cannot be reordered by CPU: Line 1-2 cannot be reordered because of the `StoreLoadFence()`, line 2-3 cannot be reordered since line 3 is a store that is only executed if line 2 is true. The two lines in GC cannot be reordered by CPU because line 2 stores to the same field `o.cellState` as line 1.

In addition, note that it's fine if at the end of `WriteBarrier`, the object is `black` but GC has only executed to line 1: this is unfortunate, because the next `WriteBarrier` on this object will add the object to the remembered set despite it being unnecessary. However, it does not affect our correctness. So now, let's enumerate all the interleavings again!

Interleaving 1.

1. [WriteBarrier] Store(o.cellState, white)
2. [WriteBarrier] t1 = isMarked() // t1 = false
3. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // not executed

Object is not marked and white, OK.

Interleaving 2.

1. [WriteBarrier] Store(o.cellState, white)
2. [WriteBarrier] t1 = isMarked() // t1 = false
3. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed 'o' into queue
4. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // not executed

Object is in queue and white, OK.

Interleaving 3.

1. [WriteBarrier] Store(o.cellState, white)
2. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed 'o' into queue
3. [WriteBarrier] t1 = isMarked() // t1 = true
4. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is in queue and black, unfortunate but OK.

Interleaving 4.

1. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed 'o' into queue
2. [WriteBarrier] Store(o.cellState, white)
3. [WriteBarrier] t1 = isMarked() // t1 = true
4. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is in queue and black, unfortunate but OK.

Interleaving 5.

1. [WriteBarrier] Store(o.cellState, white)
2. [WriteBarrier] t1 = isMarked() // t1 = false

3. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
4. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
5. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // not executed

Object is marked and black, OK.

Interleaving 6.

1. [WriteBarrier] Store(o.cellState, white)
2. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
3. [WriteBarrier] t1 = isMarked() // t1 = true
4. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
5. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is marked and black, OK.

Interleaving 7.

1. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
2. [WriteBarrier] Store(o.cellState, white)
3. [WriteBarrier] t1 = isMarked() // t1 = true
4. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
5. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is marked and black, OK.

Interleaving 8.

1. [WriteBarrier] Store(o.cellState, white)
2. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
3. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
4. [WriteBarrier] t1 = isMarked() // t1 = true
5. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is marked and black, OK.

Interleaving 9.

1. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
2. [WriteBarrier] Store(o.cellState, white)
3. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
4. [WriteBarrier] t1 = isMarked() // t1 = true
5. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is marked and black, OK.

Interleaving 10.

1. [GC Marking] CAS(o.isMarked, true), Store(o.cellState, white), pushed ‘o’ into queue
2. [GC Marking] Popped ‘o’ from queue, Store(o.cellState, black)
3. [WriteBarrier] Store(o.cellState, white)
4. [WriteBarrier] t1 = isMarked() // t1 = true
5. [WriteBarrier] if (t1 == true): Store(o.cellState, black) // executed

Object is marked and black, OK.

So let's update our pseudo-code. However, I would like to note that, in JSC's implementation, they did not use a `StoreLoadFence()` after `obj->cellState = white`. Instead, they made the `obj->cellState = white` a CAS from `black` to `white` (with memory ordering `memory_order_seq_cst`). This is stronger than a `StoreLoadFence()` so their logic is also correct. Nevertheless, just in case my analysis above missed some other race with other components, our pseudo-code will stick to their logic…

Mutator `WriteBarrier` pseudo-code:

```
void WriteBarrier(JSCell* obj) {
    StoreLoadFence(); // Note the fence!
    if (obj->cellState == black)
        WriteBarrierSlowPath(obj);
}

void WriteBarrierSlowPath(JSCell* obj) {
    if (IsGcRunning()) {
        if (!isMarked(obj)) {
            if (SUCCESS ==
                CompareAndSwap(obj->cellState, black /*from*/, white /*to*/)) {
                if (isMarked(obj)) {
                    obj->cellState = black;
                }
            }
            return;
        }
    } else {
        assert(isMarked(obj));
    }

    obj->cellState = white;
    // Add 'obj' to remembered set
    rmbSet.push(obj);
}
```

Eden/Full GC Marking phase:

```
while (!queue.empty() || !rmbSet.empty()) {
    JSCell* obj = !queue.empty() ? queue.pop() : rmbSet.pop();
    obj->cellState = black;

    StoreLoadFence(); // Note the fence!

    obj->ForEachChild([&](JSCell* child) {
        if (!child->isMarked) {
            markObject(child);
            child->cellState = white;
            queue.push(child);
        }
    });
}
```

Remove Unnecessary Memory Fence In WriteBarrier

The `WriteBarrier` is now free of hazardous race conditions. However, we are executing a `StoreLoadFence()` for every `WriteBarrier`, which is a very expensive CPU instruction. Can we optimize it?

The idea is the following: the fence is used to protect against race with GC. Therefore, we definitely need the fence if the GC is concurrently running. However, the fence is unnecessary if the GC is not

running. Therefore, we can check if the GC is running first, and only execute the fence if the GC is indeed running.

JSC is even more clever: instead of having two checks (one that checks if the GC is running and one that checks if the `cellState` is `black`), it combines them into a single check for the fast-path where the GC is not running and the object is `white`. The trick is the following:

1. Assume `black = 0` and `white = 1` in the `cellState` enum.
2. Create a global variable called `blackThreshold`. This `blackThreshold` is normally `0`, but at the beginning of a GC cycle, it will be set to `1`, and it will be reset back to `0` at the end of the GC cycle.
3. Now, check if `obj->cellState > blackThreshold`.

Then, if the check succeeded, we know we can immediately return: the only case this check can succeed is when the GC is not running and we are `white` (because `blackThreshold = 0` and `cellState = 1` is the only situation to pass the check). This way, the fast path only executes one check. If the check fails, then we fallback to the slow path, which performs the full procedure: check if GC is running, execute a fence if needed, then check if `cellState` is `black` again. In pseudo-code:

```
void WriteBarrier(JSCell* obj) {
    if (obj->cellState > g_blackThreshold) {
        // Fast-path: the only way to reach here is when
        // the GC is not running and the cellState is white
        return;
    }

    if (!IsGcRunning()) {
        // g_blackThreshold is 0, so our object is
        // actually black, we need to go to WriteBarrierSlowPath
        WriteBarrierSlowPath(obj);
    } else {
        // GC is running so we need to execute the fence
        // and check cellState again
        StoreLoadFence();
        if (obj->cellState == black) {
            WriteBarrierSlowPath(obj);
        }
    }
}
```

Note that there is no race between `WriteBarrier` and GC setting/clearing `IsGcRunning()` flag and changing the `g_blackThreshold` value, because the mutator is always stopped at a safe point (of course, halfway inside `WriteBarrier` is not a safe point) when the GC starts/finishes.

“Obstruction-Free Double Collect Snapshot”

The concurrent GC also introduced new complexities for the `ForEachChild` function used by the GC marking phase to scan all objects referenced by a certain object. Each JavaScript object has a `Structure` (aka, hidden class) that describes how the content of this object shall be interpreted into object fields. Since the GC marking phase is run concurrently with the mutator, and the mutator may change the `Structure` of the object, and may even change the size of the object’s butterfly, the GC must be sure that despite the race conditions, it will never crash by dereferencing invalid pointers and never miss to scan a child. Using a lock is clearly infeasible for performance reasons. JSC uses a so-called *obstruction-free double collect snapshot* to solve this problem. Please refer to Filip Pizlo’s GC blog post to see how it works.

Some Minor Design Details and Optimizations

You might find this section helpful if you want to actually read and understand the code of JSC, but otherwise feel free to skip it: these details are not centric to the design, and are not particularly interesting either. I mention them only to bridge the gap between the GC scheme explained in this post and the actual implementation in JSC.

As explained earlier, each `CompleteSubspace` owns a list of `BlockDirectory` to handle allocations of different sizes; each `BlockDirectory` has an active block `m_currentBlock` where it allocates from, and it achieves this by holding a free list of all available cells in the block. But how does it work exactly?

As it turns out, each `BlockDirectory` has a `cursor`, which is reset to point at the beginning of the block list at the end of an eden or full GC cycle. Until it is reset, it can only move forward. The `BlockDirectory` will move the cursor forward, until it finds a block containing available cells, and allocate from it. If the cursor reaches the end of the list, it will attempt to steal a 16KB block from another `BlockDirectory` and allocate from it. If that also fails, it will allocate a new 16KB block from malloc and allocate from it.

I also mentioned that a `BlockDirectory` uses a free list to allocate from the currently active block `m_currentBlock`. It's important to note that in the actual implementation of JSC, the cells in `m_currentBlock` does not respect the rule for `isNew` bit. Therefore, to check liveness, one either needs to do a special-case check to see if the cell is from `m_currentBlock` (for example, see HeapCell::isLive), or, for the GC[24], stop the mutator, destroy the free list (and populate `isNew` in the process), do whatever inspection, then rebuild the free list and resume the mutator. The latter is implemented by two functions named `stopAllocating()` and `resumeAllocating()`, which are automatically called whenever the world is stopped or resumed.

The motivation of allowing `m_currentBlock` to not respect the rule for `isNew` is (a tiny bit of) performance. Instead of manually setting `isNew` to `true` for every allocation, a block-level bit `allocated` (aggregated as a bitvector in `BlockDirectory`) is used to indicate if a block is full of live objects. When the free list becomes empty (i.e., the block is fully allocated), we simply set `allocated` to `true` for this block. When querying cell liveness, we check this bit first and directly return true if it is set. The `allocated` bitvector is cleared at the end of each GC cycle, and since the global logical version for `isNew` is also bumped, this effectively clears all the `isNew` bits, just as we desired.

JSC's design also supports the so-called *constraint solver*, which allows specification of implicit reference edges (i.e., edges not represented as pointer in the object). This is mainly used to support JavaScript interaction with DOM. This part is not covered in this post.

Weak references have multiple implementations in JSC. The general (but less efficient) implementation is `WeakImpl`, denoting a weak reference edge. The data structure managing them is `WeakSet`, and you can see it in every block footer, and in every PreciseAllocation GC header. However, JSC also employs more efficient specialized implementations to handle the weak map feature in JavaScript. The details are not covered in this post.

In JSC, objects may also have destructors. There are three ways destructors are run. First, when we begin allocating from a block, destructors of the dead cells are run. Second, the `IncrementalSweeper` periodically scans the blocks and runs destructors. Finally, when the VM shuts down, the `lastChanceToFinalize()` function is called to ensure that all destructors are run at that time. The details of `lastChanceToFinalize()` are not covered in this post.

JSC employs a conservative approach for pointers on the stack and in registers: the GC uses UNIX signals to suspend the mutator thread, so it can copy its stack contents and CPU register values to search for data that looks like pointers. However, it's important to note that a UNIX signal is **not** used

to suspend the execution of the mutator: the mutator always **actively** suspends itself at a safe point. This is critical, as otherwise it could be suspended at weird places, for example, in a `HeapCell::isLive` check after it has read `isNew` but before it has read `isMarked`, and then GC did `isNew |= isMarked, isMarked = false`, and boom. So it seems like the only reason to suspend the thread is for the GC to get the CPU register values, including the `SP` register value so the GC knows where the stack ends. It's unclear to me if it's possible to do so in a cooperative manner instead of using costly UNIX signals.

Acknowledgements

I thank Saam Barati from Apple's JSC team for his enormous help on this blog post. Of course, any mistakes in this post are mine.

Footnotes

1. A brief stop-the-world pause is still required at the start and end of each GC cycle, and may be intentionally performed if the mutator thread (i.e.the thread running JavaScript code) is producing garbage too fast for the GC thread to keep up with. ↩
2. The actual allocation logic is implemented in LocalAllocator. Despite that in the code `BlockDirectory` is holding a linked list of `LocalAllocator`, (at time of writing, for the codebase version linked in this blog) the linked list always contains exactly one element, so the `BlockDirectory` and `LocalAllocator` is one-to-one and can be viewed as an integrated component. This relationship might change in the future, but it doesn't matter for the purpose of this post anyway. ↩
3. Since the footer resides at the end of a 16KB block, and the block is also 16KB aligned, one can do a simple bit math from any object pointer to access the footer of the block it resides in. ↩
4. Similar to that per-cell information is aggregated and stored in the block footer, per-block information is aggregated as bitvectors and stored in `BlockDirectory` for fast lookup. Specifically, two bitvectors `empty` and `canAllocateButNotEmpty` track if a block is empty, or partially empty. The code is relatively confusing because the bitvectors are laid out in a non-standard way to make resizing easier, but conceptually it's just one bitvector for each boolean per-block property. ↩
5. While seemingly straightforward, it is not straightforward at all (as you can see in the code). The free cells are marked free by the GC, and due to concurrency and performance optimization the logic becomes very tricky: we will revisit this later. ↩
6. In fact, it also attempts to steal blocks from other allocators, and the OS memory allocator may have some special requirements required for the VM, but we ignore those details for simplicity. ↩
7. In the current implementation, the list of sizes (byte) are 16, 32, 48, 64, 80, then `80 * 1.4 ^ n` for `n >= 1` up to about 8KB. Exponential growth guarantees that the overhead due to internal fragmentation is at most a fraction (in this case, 40%) of the total allocation size. ↩
8. An interesting implementation detail is that `IsoSubspace` and `CompleteSubspace` always return memory aligned to 16 bytes, but `PreciseAllocation` always return memory address that has reminder 8 module 16. This allows identifying whether an object is allocated by `PreciseAllocation` with a simple bit math. ↩
9. JSC has another small optimization here. Sometimes a `IsoSubspace` contains so few objects that it's a waste to hold them using a 16KB memory page (the block size of `BlockDirectory`). So the first few memory pages of `IsoSubspace` use the so-called "lower-tier", which are smaller memory pages allocated by `PreciseAllocation`. In this post, we will ignore this design detail for simplicity. ↩

10. Memory of an `IsoSubspace` is only used by this `IsoSubspace`, never stolen by other allocators. As a result, a memory address in `IsoSubspace` can only be reused to allocate objects of the same type. So for any type `A` allocated by `IsoSubspace`, even if there is a use-after-free bug on type `A`, it is impossible to allocate `A`, free it, allocate type `B` at the same address, and exploit the bug to trick the VM into interpreting an integer field in `B` controlled by attacker as a pointer field in `A`. ↩
11. In some GC schemes, an eden object is required to survive two (instead of one) eden GCs to be considered in old space. The purpose of such design is to make sure that any old space object is at least one eden-GC-gap old. In contrast, in JSC's design, an object created immediately before an eden collection will be considered to be in old space immediately, which then can only be reclaimed via a full GC. The performance difference between the two designs is unclear to me. I conjecture JSC chose its current design because it's easier to make concurrent. ↩
12. There is one additional color `Grey` in the code. However, it turns out that `White` and `Grey` makes no difference (you can verify it by grepping all use of `cellState` and observe that the only comparison on `cellState` is checking if it is `Black`). The comments explaining what the colors mean do not fully capture all the invariants. In my opinion JSC should really clean it up and update the comment, as it can easily cause confusion to readers who intend to understand the design. ↩
13. The bit is actually called `isNewlyAllocated` in the code. We shorten it to `isNew` for convenience in this post. ↩
14. *Safe point* is a terminology in GC. At a *safe point*, the heap and stack is in a coherent state understandable by the GC, so the GC can correctly trace out which objects are dead or live. ↩
15. For `PreciseAllocation`, all allocated objects are chained into a linked list, so we can traverse all objects (live or dead) easily. This is not efficient: we will explain the optimizations for `CompleteSubspace` later. ↩
16. Keep in mind that while this is true for now, as we add more optimizations to the design, this will no longer be true. ↩
17. Note that we push the old space object into the queue, not the eden object, because this pointer could have been overwritten at the start of the GC cycle, making the eden object potentially collectable. ↩
18. Also note that all objects dead before this GC cycle, i.e. the free cells of a block in `CompleteSubspace`, still have `isNew = false` and `isMarked = false`, as desired. ↩
19. Recall that under generational hypothesis, most objects die young. Therefore, that "all objects in an eden block are found dead during eden GC" is something completely plausible. ↩
20. In JSC, the version is stored in a `uint32_t` and they have a bunch of logic to handle the case that it overflows `uint32_t`. In my humble opinion, this is an overoptimization that results in very hard-to-test edge cases, especially in a concurrent setting. So we will ignore this complexity: one can easily avoid these by spending 8 more bytes per block footer to have `uint64_t` version number instead. ↩
21. Note that any number of eden GC cycles may have run between the last full GC cycle and the current full GC cycle, but eden GC does not bump mark version. So for any object born before the last GC cycle (no matter eden or full), the `isMarked` bit honestly reflects if it is live, and we will accept the bit as its mark version must be off-by-one. For objects born after the last GC cycle, it must have a latest `isNew` version, so we can know it's alive through `isNew`. In both cases, the scheme correctly determines if an object is alive, just as desired. ↩
22. And probably not: first, true sharing and false sharing between GC and mutator can cause slowdowns. Second, as we have covered before, JSC uses a Time-Space Scheduler to prevent the mutator from allocating too fast while the GC is running. Specifically, the

mutator will be intentionally suspended for at least 30% of the duration. So as long as the GC is running, the mutator suffers from an 30%-or-more “performance tax”. ↩

23. The real story is a bit more complicated. JSC actually reuse the same VM for different JavaScript scripts. However, at any moment, at most one of the script can be running. So technically, there are multiple mutually-exclusive mutator threads, but this doesn’t affect our GC story. ↩
24. The GC needs to inspect a lot of cells, and its logic is already complex enough, so having one less special-case branch is probably beneficial for both engineering and performance. ↩