Concurrent JavaScript: It can work!

Aug 30, 2017

by Filip Pizlo

@filpizlo

With the recent addition of `SharedArrayBuffer`, concurrency is finding its way into the JavaScript language. This addition allows JavaScript programs to perform concurrent access to `SharedArrayBuffer` objects. WebKit supports `SharedArrayBuffer` and it has full optimization support in our compiler pipeline. Unfortunately, JavaScript does not allow any objects other than `SharedArrayBuffer` to be shared.

This post considers a wild thought experiment: what would it take to extend concurrency to the entire JavaScript heap? In this world, any object can be shared with other threads. This would be no small change. Existing JavaScript VM optimizations exploit the fact that there is only one thread of execution, so concurrency is sure to introduce some performance problems. This post is concerned with whether this is even technically feasible, and if it is, what the cost might be.

We offer a basic strawman API to illustrate what we mean by concurrency. Most of the post is concerned with how WebKit's JavaScript VM (called JavaScriptCore, or JSC for short) can implement the strawman. Implementing this strawman will be a big effort. We think that our proposed implementation ought to be able to meet the following goals:

- No performance regressions for code that does not use concurrency.

- Linear scalability when a program is run in parallel with itself without deliberately sharing any objects. We don't expect two threads to be fully twice as fast as one, since we expect some concurrency overhead — but if you have two CPUs then two threads should be faster than one, and hopefully close to twice as fast as one.
- Linear scalability — including speed-ups versus the best serial baselines — on some corpus of parallel code that does share objects.
- Sensible semantics, including a memory model that isn't any weaker than what modern hardware provides.
- Compatibility. For example, concurrent JavaScript programs should have a story for how to use the DOM without requiring the DOM implementation to be rewritten.

Our proposed implementation scheme relies on 64-bit systems, but that is largely because our engine is already 64-bit-centric.

It's not yet possible to evaluate the performance of this scheme empirically, but our implementation sketch does make it possible to intuit what performance might look like. Specifically, our scheme ensures that most property accesses will experience at most one arithmetic instruction worth of overhead (which is almost free), with a few particularly tricky (and rare) ones experiencing up to about 7x overhead. This post ends with a section that compares our scheme to other techniques for implementing concurrency.

Strawman Concurrent JS

Our strawman proposal for concurrent JS is to simply add threads. Threads gets separate stacks but share everything else. Threads are great for our experiment because they are so general. We can imagine implementing many other kinds of concurrent programming models on top of threads. Thus, if we can get our VM to support threads, then we can probably get it to support lots of other concurrent and parallel programming models. This post is about removing

technical feasibility as a gating factor for adding any concurrent programming model to JavaScript.

This section makes the strawman somewhat concrete, mostly to provide context on the aspects of it that are easy or hard to implement. It's useful to know what the API looks like in order to understand what constraints it creates.

Each program will start out with one thread. Threads can start other threads. Threads live in the same heap and can share objects with each other. This section shows the API, describes the memory model, and shows how concurrency interacts with the DOM.

Possible API

We propose:

- a simple API for creating threads,
- a change to the `Atomics` object to support building lock objects,
- a lock and condition variable API that can be built on top of Atomics,
- a way to create thread-local variables, and
- some helpers to allow for incremental adoption.

This post will use this strawman to understand exactly what it would take to implement threads in JavaScriptCore.

We want it to be easy to create threads:

```
new Thread(function() { console.log("Hello, threads!
```

This will start a new thread, which will eventually print "Hello, threads!". Note that even this simple example shares lots of stuff with the thread. For example, the function object captures the lexical scope in the thread in which it was created, so that when the thread accesses the `console` variable, this is an access to a shared object.

Threads can be *joined* to wait for them to finish and to get their result:

```
let result = new Thread(() => 42).join(); // returns
```

In web browsers, the main thread cannot block, so `join()` will throw an exception if the above code was on the main thread. We can support asynchronous versions of blocking operations:

```
new Thread(() => 42).asyncJoin().then((result) => /*
```

You can always get the Thread object for the current thread:

```
let myThread = Thread.current;
```

Threads may need to wait on each other to prevent races. This can be accomplished using locks. Rather than simply introduce a locking API, we propose to extend the Atomics API to allow users to build any locks they like. We provide a good lock implementation in our proposed API, but we want to encourage the creation of other kinds of synchronization primitives. The existing `SharedArrayBuffer` specification allows developers to create custom locks using the `Atomics` API. This API allows you to say things like:

```
Atomics.wait(array, index, expectedValue);
```

and:

```
Atomics.wake(array, index, numThreadsToWake);
```

Currently, the array must be an integer typed array backed by a `SharedArrayBuffer`. We propose extending all `Atomics` methods that take an array/index to take an object and a property name instead. Since an index is a property name, this does not change behavior of code that already uses this API for `SharedArrayBuffer`. This also implies that `Atomics` methods that currently take integer values (for storing or comparing to elements in typed arrays) will now be able to take any JavaScript value when used with normal JavaScript properties. `Atomics.wake`, `Atomics.wait`, and `Atomics.compareExchange` are sufficient for implementing any a lock using just one JavaScript property.

Additionally, we propose adding a Lock API:

```
let lock = new Lock();
lock.hold(function() { /* ...perform work with lock
```

Locking on the main thread is possible with promises:

```
lock.asyncHold().then(function() { /* ...perform wor
```

This works because each thread gets its own runloop. We can also add a Condition API:

```
let cond = new Condition();
cond.wait(lock); // Wait for a notification while th
// ...
```

```
cond.asyncWait(lock).then(function() { /* ...perform
// ...
cond.notify(); // Notify one thread or promise.
// ...
cond.notifyAll(); // Notify all threads and promises
```

`Condition.prototype.wait` will release the lock you pass it before waiting, and reacquire it before returning. The async variant associates the resulting promise with the condition variable such that if the condition is notified, the promise will be fulfilled on the current thread.

Using `Thread.current` and `WeakMap`, anyone can implement thread-local variables. Nonetheless, it's sometimes possible for the underlying runtime to do something more clever. We don't want to require all JavaScript programmers to know how to implement thread-local variables with `WeakMap`. Hence, we propose a simple API:

```
let threadLocal = new ThreadLocal();
function foo()
{
    return threadLocal.value;
}
new Thread(function() {
    threadLocal.value = 43;
    print("Thread sees " + foo()); // Will always pr
});
threadLocal.value = 42;
print("Main thread sees " + foo()); // Will always p
```

Finally, we want to make it easy for users to assert that objects stay on one thread:

```
var o = {f: "hello"}; // Could be any object.
Thread.restrict(o);
new Thread(function() {
    console.log(o.f); // Throws ConcurrentAccessErro
});
```

Any object that is `Thread.restrict`ed should throw `ConcurrencyAccessError` in response to any proxyable operation performed by a thread other than the one that called `Thread.restrict`.

Memory Model

Processors and compilers like to reorder memory accesses. Both processors and compilers love to *hoist* loads from memory. This happens because the following code:

```
let x = o.f
o.g = 42
let y = o.f
```

May get transformed by the compiler or processor to:

```
let x = o.f
o.g = 42
let y = x
```

Effectively, this means that the "load" into `y` got moved above the store to `o.g`. Processors will do the same optimization dynamically, by caching the load of `o.f` and reusing the cached result for the second load.

Processors and compilers like to *sink* stores to memory. This happens because the following code:

```
o.f = 42
let tmp = o.g
o.f = 43
```

May get transformed by the compiler or processor to:

```
let tmp = o.g
o.f = 43
```

This is "as if" the `o.f = 42` statement got moved to just before `o.f = 43`. Even if the compiler does not perform this transformation, the processor may buffer stores and execute them when it is convenient. This can also mean that `o.f = 42` may execute after `let tmp = o.g`.

It makes most sense for our strawman to follow the existing `SharedArrayBuffer` memory model. It's already possible to write multithreaded code that has boundedly nondeterministic behavior using `SharedArrayBuffer`, and we aren't going to protect completely from such code. But JavaScript's objects are much more complicated than a buffer, so guaranteeing that basic invariants of the JS object model hold in the face of races is nontrivial. We propose that operations that modify JavaScript object storage execute atomically. Atomicity means that if many threads execute these operations concurrently, then each one will behave as if they had all executed in some sequence with no concurrency. Each of the underlying operations that JS can do to objects should be atomic:

- Add a property.
- Delete a property.
- Get a property's value.
- Set a property's value.
- Change a property's configuration.
- Snapshot the set of property names.

This doesn't always mean that the expression `o.f` is atomic, since this expression may do much more than loading a property's value. In particular:

- If `o.f` is a plain property directly on `o`, then it is atomic.
- If `o.f` is a prototype access, then loading the prototype is separate from loading `f` from the prototype.
- If `o.f` is a getter, then loading the getter is one step (which would be atomic if the getter is directly on `o`) but calling the getter is not atomic, since the getter may execute arbitrary code.

We propose that the low-level object operations are atomic. Some operations, like getting and setting a property's value, may be implemented using hardware primitives that allow reordering. We propose to allow reorderings of get/set accesses around each other subject to the same memory model for get/set accesses to SharedArrayBuffer. While our strawman does allow for races and some memory model strangeness, it does not allow for JavaScript's object model invariants to be invalidated. For any heap created by a concurrent JS program, it should be possible to write a sequential JS program that creates an indistinguishable heap.

Finally, we propose that memory management of the concurrent JS heap happens just as it does in other garbage-collected multi-threaded languages. Quite simply, garbage collection must appear to happen atomically and only at well defined safepoints, like loop back edges, allocation sites, and during calls to native code that doesn't touch the heap. This requirement is satisfied by popular garbage collection algorithms like those in HotSpot or MMTk. It's also satisfied by classic algorithms like mark-sweep and semi-space, and even in garbage collectors with no global stop-the-world phase, including ones that are so lock-free that they don't even stop threads to scan their stacks. WebKit's Riptide GC already mostly supports multiple threads because our JIT threads can access the heap.

Interaction With The DOM

Extending concurrency to all of JavaScript will be hard; extending it to all of the DOM will be even harder. We extend enough reasoning about threads to the DOM to make this strawman useful.

We propose that by default, DOM objects throw `ConcurrentAccessError` in response to any proxyable operation from threads other than the main one, just as if `Thread.restrict` had been called on them from the main thread.

However, a few objects will have to allow for concurrent accesses just to make the language behave sanely. Something obvious like:

```
new Thread(function() { console.log("Hello, threads!
```

requires a concurrent access to a DOM object. In WebKit, *DOM global objects* are in charge of storing the variable state of a `window`. Ordinary JS properties of the global object (including `console`, `Object`, etc) have to be accessible to concurrent threads, because those properties are the scripts's global variables. Accesses from other threads to exotic properties of the global object, or properties of the global object's prototype, can throw. Attempts to add new properties to the global object or delete properties from it from non-main threads can throw as well. These restrictions mean that in WebKit, handling the DOM global object's limited kind of concurrent property accesses is not much harder than handling those same accesses for ordinary JS objects.

Additionally, namespace objects like `console` can be accessed by concurrent threads because they can just use the JS object model. There is no reason to restrict them to the main thread, and it's important that `console` is accessible due to its utility for debugging.

Strawman Summary

This section has proposed a strawman API for threading in JavaScript. This API is powerful enough to implement custom synchronization primitives. It's also powerful enough to allow for the execution of racy programs, but it does not allow races to break the language. Threads are general enough to allow for many other kinds of programming models to be implemented on top of them.

Implementing Concurrent JS in WebKit

This section shows that we can implement our threads-based strawman without having to disable any of JavaScriptCore's fundamental performance optimizations. Our scheme aims to achieve close to zero overhead even in the case of programs that read and write to the same objects from multiple threads.

JavaScript shares a lot in common with languages like Java and .NET, which already support threads. Here are some of the things these languages have in common with JavaScript:

- Like JavaScript, those languages use tracing-based garbage collectors. Our GC mostly supports multiple threads, since JIT threads are already allowed to read the heap. The biggest missing piece is thread local allocation, which enables concurrent allocation. For our GC this just means a separate `FreeList` per thread for each allocator. Our GC has a fixed number of allocators and we already have fast thread-local storage, so this will be a mechanical change.
- Like JavaScript, those languages are implemented using multiple tiers of JITs and possibly an interpreter. Our WebAssembly VM already supports multi-threaded tier-up from BBQ (build bytecode quickly) to OMG (optimized machinecode generation). We're not worried about getting this right for JavaScript.
- Like implementations of JavaScript, implementations of those languages use inline caching to accelerate dynamic operations. We will have to make some

> changes to our inline caches to support concurrency, and we describe those changes in a later section in this post. It's not the hardest part about adding concurrency, since it's not new territory — it's been done before.

These similarities suggest that many of the techniques that are already used to make Java virtual machines support concurrency can be reused to make JavaScript concurrent.

Where things get hard is JavaScript's ability to dynamically reconfigure objects. While Java and .NET have fixed-size objects (once allocated, an object does not change size), JavaScript objects tend to be variable-size. Concurrency in those statically-typed languages relies on the fact that concurrent accesses to fixed-size objects are *atomic by default* up to the machine's pointer width (so, 64-bit systems do 64-bit property accesses atomically by default). Pointer values in Java and .NET are addresses to the contiguous slab of memory holding the object's data, and it only takes some address arithmetic (like adding an offset) and a single memory access instruction to read/write any field. Even if the memory model allows surprising reorderings, it's never the case that a racing access instruction to the same field (or different fields of the same object) corrupts the entire object or causes a crash. JavaScript's variable-size objects, on the other hand, mean that object accesses require multiple memory access instructions in some cases. A sequence of operations involving multiple accesses to memory is not atomic by default. In the worst case, the VM might crash because the internal object state gets corrupted. Even if this doesn't happen, races might cause writes to be lost or for time-travel to occur (writing A and then B to a field may lead to reads of that field to first see A, then B, and then A again).

In our strawman proposal, concurrent JavaScript implies that fundamental operations such as adding a new property to an object, changing the value of a property, reading a property, and removing a property all proceed atomically. No race should ever lead to a VM crash, lost writes, or the value of a property experiencing time travel.

We propose an algorithm that allows most JavaScript object accesses to be *wait-free* and require minimal overhead compared to our existing serial JS implementation. Wait-free operations execute without ever blocking and complete in a bounded number of steps regardless of contention. This algorithm borrows ideas from real-time garbage collection, locking algorithms, and type inference. We plan to use a tiered defense against the overheads of concurrency:

1. We propose using our existing polymorphic-inline-cache-based type inference system to infer relaxed notions of thread-locality of objects (and their types) that we call *transition-thread-locality* (TTL). TTL objects will use the same object model as they do today and accesses to those objects will enjoy close to zero overhead. TTL does not imply true thread-locality; for example even objects that are read and written by many threads may be inferred TTL.
2. We propose using a *segmented* object model for objects that fail TTL inference. This will introduce an extra load instruction, and some arithmetic, to the property access fast path. We fear that by itself, this technique would not be fast enough to meet our performance goals — but since the segmented object model will only be used surgically for those objects (or types of objects) that fail TTL inference, we suspect that the *net* cost will be small enough to be practical.
3. Operations on non-TTL objects that don't benefit from the segmented object model will use locks. During our development of the Riptide concurrent GC, we added an internal lock to each object. This lock occupies just 2 bits and allows the lock/unlock fast path to require just an atomic compare-and-swap (CAS) and a branch.

Before diving into the details, we first set up some expectations for the hardware that this proposed system would run on. Then we describe this proposal in reverse. We consider the cost of just using per-object locking. Then we review the existing JSC object model and describe what kinds of operations happen to already be atomic within this model. Next we introduce *segmented butterflies*, which are the key to allow dynamically reconfiguring objects. Then we

show how our TTL inference will allow us to use our existing object model for hopefully most of the JavaScript heap. This section then considers some loose ends, like how to do inline caching concurrently, how resizable arrays fit into TTL and segmented butterflies, how to handle large thread counts using the *local optimizer lock* (LOL), and finally how to handle native state that isn't thread-safe.

Hardware Expectations

This post describes how to convert JavaScriptCore to support concurrent JavaScript. JSC is currently optimized for 64-bit systems, and most of our existing concurrency support (concurrent JIT, concurrent GC) only works on 64-bit systems. This section briefly summarizes what we expect from a 64-bit system to be able to use our scheme.

JSC is optimized for x86-64 and ARM64. This scheme assumes the weaker of the two memory models (ARM64) and assumes that the following atomic instructions are inexpensive enough that it's practical to use them as an optimization over locking:

- 64-bit loads and stores are atomic by default. We expect that these accesses may be reordered around other accesses. But we also expect that if memory access instruction B has a dataflow dependency on memory access instruction A, then A will always come before B. For example, in a dependent load chain like `a->f->g`, `a->f` will always execute before `_->g`.
- 64-bit CAS (compare-and-swap). JSC uses 64-bit words for JavaScript properties and two important meta-data fields in the object header: the *type header* and the *butterfly pointer*. We want to be able to atomically CAS any JS property and either of the meta-data properties in the object header.
- 128-bit DCAS (double-word compare-and-swap). Sometimes, we will want to CAS all of the meta-data inside JS objects, which means CASing both the 64-bit type header and the adjacent 64-bit butterfly pointer.

Per-Object Locking

Each object in JSC already has a lock, which we use to synchronize certain fundamental JavaScript operations with the garbage collector. We can use this lock to protect any operations that need to be synchronized between JavaScript threads. The rest of this post deals with optimizations that allow us to avoid this lock. But let's consider exactly what the cost of the lock is. Our internal object lock algorithm requires an atomic compare-and-swap (CAS) and a branch for locking, and another CAS and branch for unlocking. In the best case, CAS is like executing at least 10 cycles. This is the amortized cost assuming the CAS succeeds. On some hardware, the cost is much higher. Assuming a branch is one cycle, this means at least 22 extra cycles for each object operation that requires the lock. Some rare operations are already expensive enough that 22 cycles is not bad, but many fast path operations would not be able to handle such overhead. Below we consider some operations and how much they would be affected if they had to use locking.

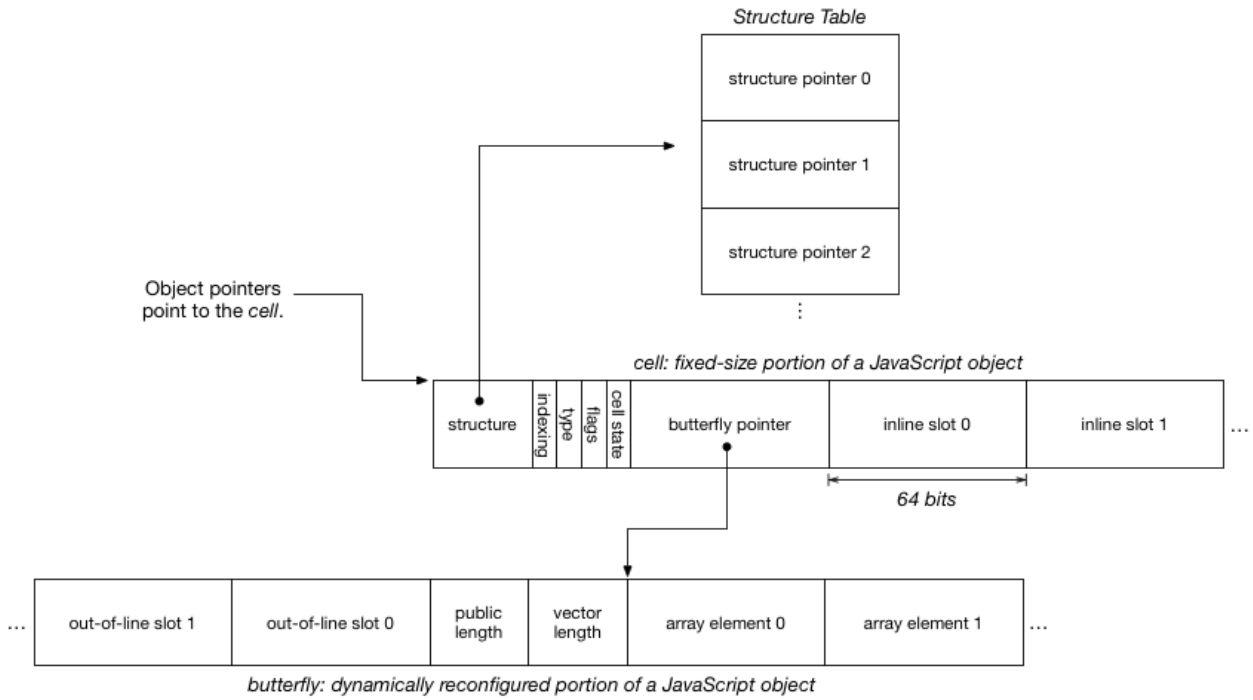
- Adding a new property currently requires only one load, a branch, and two stores on the optimized fast path. Each of those operations is a single cycle, leading to about 4 cycles total in the fastest case. Adding locking bloats this to 26 cycles – a ~7x slow-down. In some cases, adding a new property can be optimized down to just a single store. In those cases, adding locking creates a 23x regression.
- Changing the value of an existing property requires a load, a branch, and a store. That's 3 cycles in the fastest case. Adding two CASes and two branches bloats this to 25 cycles – a 8x slow-down. In some cases, we can optimize a property store to just one store instruction. Putting locking around it is a 23x regression.
- Loading an existing property's value requires a load, a branch, and another load. Adding locking is a 8x slow-down, or 23x in the case where we optimized the load to just a single load instruction.
- Deleting a property is already slow. We are fine with putting locks around it. Deleting properties is relatively rare, but we have to support it.

- Dictionary lookups — JSC-speak for property accesses that we execute dynamically without any inline cache or compiler smarts — will see a small overhead from locking. Those code paths are already expensive enough that adding locking to them is unlikely to be a problem.
- Snapshotting the set of properties does not change. It's already possible for concurrent threads to snapshot the set of properties of any object in JSC because of locking that we implemented for concurrent garbage collection.

While some operations like deletion are already expensive enough that locking is not a problem, we don't believe it's practical to add such extreme costs to the fast cases of JavaScript object access. The resulting programming language would be too slow.

Designing a fast concurrent JS implementation requires introducing new algorithms for property accesses that can run concurrently to each other without requiring any locking except in rare cases. In the sections that follow, we describe such an algorithm. First we review JavaScriptCore's object model. Then we show cases where JavaScriptCore's object model already behaves as if it had fixed-size objects; those cases never require locking. Next we show a technique called *segmented* butterflies, which allows for mostly wait-free concurrent object access. By itself, this technique is still too expensive for our tastes. So, we show how to infer which object types are *transition-thread-local* (TTL) to avoid synchronization when transitioning objects. This allows us to use flat butterflies (our existing object model) for the majority of objects. Then we describe how to handle deletion, dictionary lookups, inline caches, arrays, and thread-unsafe objects.

JavaScriptCore Object Model



The JavaScriptCore Object Model. JavaScript values and object pointers are implemented as pointers to the cell. The cell never moves and never changes size, but contains a pointer to the butterfly, which can grow either left (for named properties) or right (for array elements).

JavaScriptCore's object model allows for four kinds of state, each of which is optional:

- Native state represented using ordinary C++ objects.
- Data for named object properties (like `o.f`) that we statically guessed the object would have.
- Data for named object properties that were added dynamically, forcing us to change the size of the object.
- Data for indexed object properties. These can change the size of the object in many ways.

The first two kinds of state does not involve resizing the object. The last two kinds of state require resizing. In JSC, fixed-sized state is stored directly in the object's *cell*. The cell is what a pointer to an object points to. Within the cell, there is a *butterfly pointer* that can be used to store dynamically-allocated and resizable state in a *butterfly*. Butterflies store named properties to the left of where the butterfly pointer points inside *out-of-line slots*, and indexed

properties to the right as *array elements*. Each of those locations may store a tagged JavaScript value, which may be a number, pointer to another cell (representing a string, symbol, or object), or a special value (`true` , `false` , `null` , or `undefined`).

Every object has a *structure*, which contains the hashtable used for mapping property names to slots in the object. Objects typically share a structure with other objects that look like it (that have the same properties in the same order, among other requirements). The structure is referenced using a 32-bit index into a structure table. We do this to save space. The *indexing* and *cell state* bytes have some spare bits. We used two spare bits in the *indexing* byte to support per-object locking. We also have spare bits in the *butterfly pointer*, since pointers don't need all 64 bits on a 64-bit system.

Butterflies are optional. Many objects don't have a butterfly. When a butterfly is allocated, each of the two sides are optional. The left side of the butterfly holds the out-of-line slots. It's possible just to allocate that; in this case the butterfly pointer will point 8 bytes to the right of the end of the butterfly's memory. The right side of the butterfly holds the array elements and the array header. The *public length* is the length as it is reported by `array.length` , while the *vector length* is the number of array element slots that were allocated.

Freebies

Within this object model, accesses to anything in the cell itself are atomic by default, just like accesses to objects in Java are atomic by default. This is a big deal, since our optimizer is generally successful at putting the most important object fields inline in the cell. Concurrent JS programs that mostly rely on data that ends up inside the cell will experience almost no overhead versus their serial equivalents.

Additionally, if we know that the butterfly won't be reallocated ever again (i.e. the butterfly pointer is

immutable), then we can access it directly without any problems. Those accesses will be atomic by default.

Alternatively, if we know that only the current thread will ever resize the butterfly and all other threads will only read it, then accesses to the butterfly will be atomic by default.

Problems happen when one thread attempts to *transition* the object (i.e. add a property and/or reconfigure the butterfly) while some other thread is writing to it (or also transitioning it). If we used our current implementation of object accesses in a concurrent setting, a transition on one thread could cause races that lead to behavior that does not conform to our proposed specification.

- Writes made by another thread could disappear or experience time travel. This race happens because the transitioning thread might first complete a copy of the butterfly and then transition the object, while another thread does some more writes to the object in between those two steps. If a third thread is observing what is going on by repeatedly reading the state of the fields being written, then it will first see the state before any writes, then it will see the writes, and then once the transition completes it will see the original state again.
- Two transitions performed concurrently could cause the butterfly pointer and the object's type to be mismatched. The butterfly's format is determined by fields in the object's header that are not inside the same 64-bit word as the butterfly pointer. We cannot allow the butterfly and the header to get out of sync, as this would lead to memory corruption.
- Two transitions performed concurrently can cause some minor heap corruption even if the butterfly is not involved. If a transitions involving an inline property proceeds as two steps — first, change the object's type, and then, store the new property — then two racing transitions adding two different properties may result in one property being added, where its value ends up being the intended value of the other property. For example, a race between `o.f = 1` and `o.g = 2` could result in `o.f == 2` and `"g" in o == false`.

In the next section we show how to create an object model that has no such races, but comes with some cost. After that, we show how to use TTL inference to use our existing object model most of the time even in programs that share objects between threads.

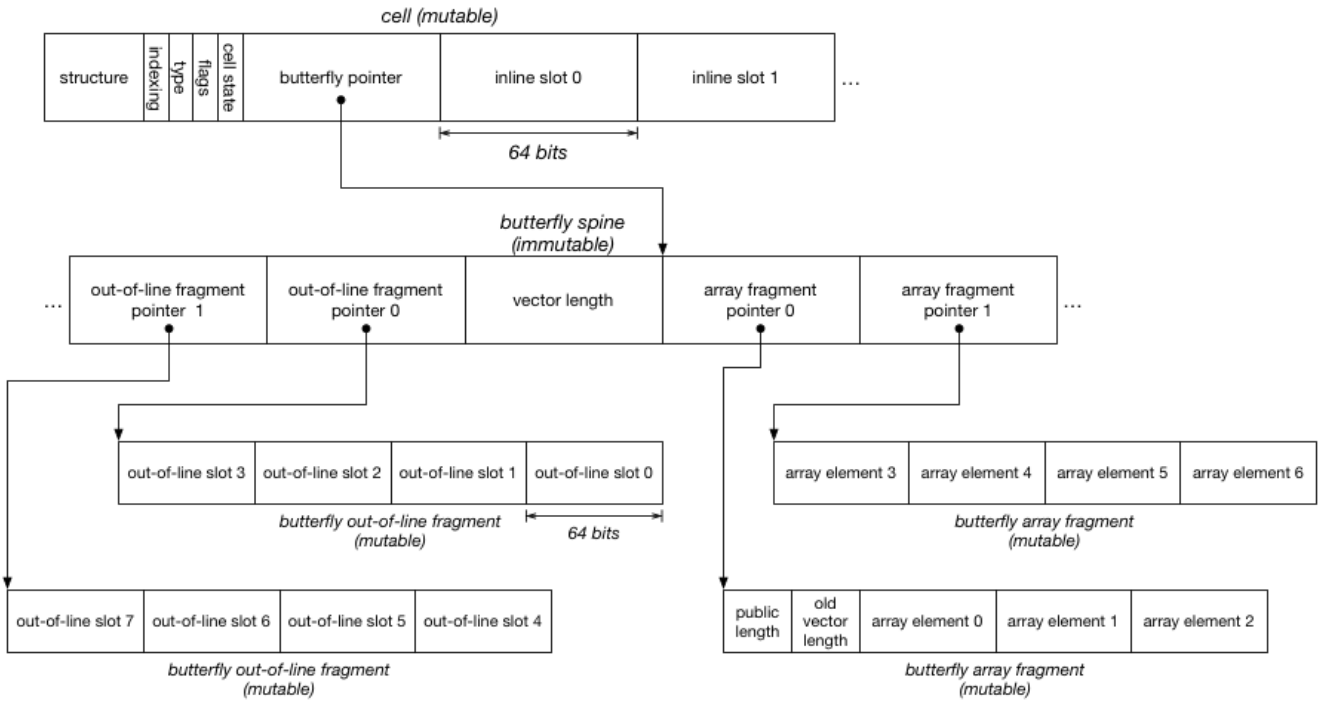
Segmented Butterflies

Transitioning an object means allocating a new butterfly, and copying the contents of the old butterfly into the new one while other threads may be reading or writing the old butterfly. We want the copy to seem as if it had happened in one atomic step. Lots of research has gone into supporting concurrent copying of objects in the context of real-time garbage collectors. The approach that we propose to use is based on the Schism real-time garbage collector's arraylet object model. This section reviews the Schism arraylet object model and then shows how it can be used for butterfly transitions.

Schism was trying to solve the problem of implementing a copying garbage collector whose copying phase was completely concurrent to the application. It was based on the observation that copying an object concurrently to other threads accessing the object is easy if the object is immutable. It's only hard if the object can be written to by the other threads. Schism solves the problem of copying *mutable* objects concurrently by boxing the mutable state in small fixed-size *fragments* (32 bytes in the paper). The object that gets copied is the *spine* that serves as an index for finding fragments. All object accesses use an extra indirection to find the fragment that contains the data being accessed. Spines are immutable objects because the fragments they point to never move.

WebKit already uses something like arraylets as well, in the `WTF::SegmentedVector` class template. We use it to prevent having to move C++ objects when the vector resizes. We also use it for implementing the `JSGlobalObject`'s variable storage (for `var` statements in global scope). The term *arraylets* is due to Bacon, Cheng, and Rajan, who used them to control fragmentation in the

Metronome garbage collector. Lots of arraylet research shows that the extra indirection has a high cost (often 10% or more). That is, if you double the cost of each array access by adding an extra indirection, then you will increase the total run time of the program by 10%.



Segmented Butterflies. A segmented butterfly comprises a spine (which resizes but contains only immutable state) and zero or more fragments (which don't resize but are mutable). Splitting resizable state from mutable state to enable concurrent layout changes is a proven technique in concurrent garbage collection. The shape of the fragments matches the shape of an unsegmented butterfly, which we will use for some additional optimizations.

We can use this trick for butterflies. A *segmented butterfly* has a spine that contains pointers to fragments that contain mutable data. If we need to add new properties and those properties won't fit in an existing fragment, we can grow the spine and allocate more fragments. Growing the spine implies reallocating it. We can safely allocate a new spine and `memcpy` the contents of the old one into it, since the spine's contents never changes. In this world, no writes ever get lost or experience time travel when a butterfly is being transitioned. An access that races with the transition either gets to a fragment using the old spine, or the new spine; since both spines will contain the same fragment addresses for properties that existed before the transition, every

possible interleaving results in threads reading and writing the same fragments.

The most natural way to implement segmented butterflies is probably to have 32-byte fragments like in Schism, since this is also a sweet-spot for our GC. The *vector length* is be inside the butterfly spine, since this is an immutable property of that spine. The vector length allows the spine to self-identify how big its right side is. The *public length* is mutable, so we want to put it into one of the fragments. Note that the first fragment also contains the *old vector length*. This property is unused when we are using segmented butterflies. It allows us to convert a flat butterfly into a segmented butterfly by having the spine point at slices of the flat butterfly; we discuss this optimization more in a later section.

This still leaves the problem of concurrent transitions. Transitions have to reallocate the spine, change some data in the type header, and store a property's value. Reallocating the spine is optional since usually some fragment will already have a spare slot. Changing the type is optional, for example in case of array resizing. The type header and butterfly can be set atomically using a DCAS (double-world compare-and-swap; 64-bit systems typically support 128-bit CAS). But this isn't enough, since no matter if we set the property's value before or after the DCAS, we will have a race. The property's slot can be anywhere in memory, so there's no way to use CAS to simultaneously do the whole transition.

If we set the newly-added value slot before changing everything else, we risk a race in which one thread tries to use some object slot for adding field `f` while another thread tries to use the same slot for field `g`. If we're not careful, the second thread might win the race for the type (so everyone thinks we added property `g`) while the first thread wins the race for the property (so we get thread 1's intended value for property `f` appearing as if it was stored by thread 2 into `g`).

If we set the newly-added value slot after changing everything else, then all loads have to contend with the possibility that slots can have “holes”. In other words, at any time, an object may claim a type that includes a property `f` even though the object does not have a value for `f` yet. We could change the language to allow this: putting a new property into an object would first involve defining it to `undefined` and then storing the real value.

We choose to put a lock around transitions. This lock isn’t for guarding races between transitions and other accesses, since those are atomic by default thanks to segmented butterflies — it’s for protecting transitions from each other. To ensure that objects don’t have holes, transitions store the new property’s value before changing the type. Transitions can use this algorithm:

1. Allocate whatever memory needs to be allocated.
2. Acquire the lock.
3. Determine if we allocated the right amount of memory; if not, release the lock and go back to step 1.
4. Store the new property’s value.
5. Change the type and butterfly.
6. Release the lock.

This results in the following cost model:

- Transitions are about 7x costlier, since they now require locking.
- Reading or writing existing property slots requires an extra load.
- Deleting and dictionary lookups still work (we show details in a later section).

Recall that the arraylet research shows 10% or higher costs if *arrays* use a segmented object model. We’re proposing to use it for ordinary properties as well, if those properties were added in a sufficiently subtle way that we weren’t able to inline them into the cell. It’s probably safe to assume that if we implemented literally this, we incur at least a 10% slow-

down. We would like to have close to zero overhead. The next section shows how we think we can get there.

Transition Thread Locality and Flat Butterflies

Segmented butterflies are only necessary when:

- Some thread other than the thread that allocated an object tries to transition the object.
- The thread that allocated the object tries to transition it, and other threads may be writing to it.

We can use flat butterflies — our existing object model — when we know that these things haven't happened yet. This section describes a hybrid object model that uses either flat or segmented butterflies, depending on whether or not we have detected a possible write-transition race. This object model also allows us to avoid locking while doing many transitions.

On a 64-bit system, butterfly pointers have 48 bits of pointer information and have zeroes in their high 16 bits. 16 bits is enough to store:

- The ID of the thread that allocated the object. We call this the butterfly *TID*.
- A bit to indicate if any thread other than the allocating thread has attempted to write to the butterfly. We call this the butterfly *shared-write* (SW) bit.

Some systems have more than 48 pointer bits. In a later section, we show how to make this scheme useful even if we had to use fewer bits. Also, we could restrict butterfly allocations to the lower 2^{48} addresses if we really wanted 16 spare bits.

Let's say that these bits are encoded as follows:

```
static const uint16_t mainThreadTID = 0; // Occassio
static const uint16_t notTTLTID = 0x7fff; // For whe
```

```
static inline uint64_t encodeButterflyHeader(uint16_
{
    ASSERT(tid <= notTTLTID); // Only support 2^15 t
    return (static_cast<uint64_t>(tid) << 48)
         | (static_cast<uint64_t>(sharedWrite) << 63
}

static inline uint64_t encodeButterfly(Butterfly* bu
{
    return static_cast<uint64_t>(bitwise_cast<uintpt
         | encodeButterflyHeader(tid, sharedWrite);
}
```

We can use flat butterflies — our existing object model — whenever the TID matches the current thread. We set the SW bit and use a magic TID value (all bits set) to indicate that the butterfly is segmented. This section shows how to use these bits to allow the use of flat butterflies anytime we know that no transition race can occur. We call this *transition thread locality inference*.

Anytime a thread attempts to write to a butterfly and the TID matches the current thread, it can simply write to the butterfly without doing anything special.

Anytime a thread attempts to write to a butterfly, the TID does not match, but the SW bit is set, it can simply write to the butterfly as well.

Anytime a thread attempts to read a butterfly, it just needs to check that the TID to determine if the read should be segmented (extra indirection) or not.

Anytime a read or write encounters TID = notTTLTID and SW = true, it knows to use the segmented object model.

The following cases need special handling:

- A thread other than the one identified by the TID attempts to write to the butterfly and the SW bit is not yet set. In this case, it needs to set the SW bit. This does not mean that the butterfly needs to become segmented. It only means that the SW bit must be set, so that any future attempts to transition the butterfly from the owning thread triggers a transition to the segmented butterfly object model.
- A thread other than the one identified by the TID attempts to transition the butterfly. In this case, it needs to set the SW bit and all TID bits and perform a segmented butterfly transition. We describe this process in detail below.
- A thread attempts to transition a butterfly that has the SW bit set. This also necessitates a segmented butterfly transition.
- Even transitions with TID = current and SW = false need locking, to make sure that those assumptions aren't violated during the transition.

The sections that follow show further refinements that reduce overhead even more. First we consider how to avoid checking TID and SW bits on every operation. Next we show how to use structure watchpointing to avoid locking on the most common transitions. Then we show how to handle array resizing. Finally we explain in more detail how flat butterflies can be transitioned to segmented ones.

Quickly Checking TID And SW Bits

This section will show how to make concurrent JS as fast as serial JS in many of the most important cases by simply extending optimizations that JavaScriptCore already uses.

JSC optimizes the code for heap accesses using *inline caching*. We use inline caches for accesses to named properties (like `o.f`) and for array accesses (like `a[i]`).

Objects that have the same properties at the same offsets will usually share the same *structure*, which is identified by 32 bits in the object's type header. If a property access tends to see the same structure, then we will generate new

machine code for this property access, which checks that the object has the expected structure and then accesses the property directly. Failed speculation causes recompilation of the inline cache. If the inline cache remains stable (doesn't recompile much) for a long time and the function that contains it becomes eligible for optimized JIT compilation, then the optimizing JIT compiler might express the inline cache's code directly in its IR, which has two outcomes: the structure check branches to OSR exit if it fails (causing abrupt termination of the optimized code's execution), and all of the code for the inline cache (the structure check, the memory access, and any other steps) become eligible for low-level optimization by our DFG and B3 JIT compiler pipelines. Our compilers are good at making type-stable JavaScript property accesses perform great.

Array accesses use similar techniques to detect if an object has array elements, and if so, how they are formatted. We support multiple kinds of storage for array elements depending on how they are being used. Inline caches detect what kind of array each array access is accessing, and we then emit code that speculates for that kind of array access.

Another technique we already use is virtual memory. For example, our WebAssembly implementation uses virtual memory tricks to check the bounds of linear memory accesses for free. We can catch page faults using POSIX signal handlers or Mach exceptions. The handler will know the exact machine state at the point of the fault, and has the power to transfer execution to any state it likes. WebAssembly uses this to throw an exception, but we can use it to transfer control flow to slow paths that handle the generic case of a memory access. Essentially, this means that we can save cycles on safety checks if we can express the condition being checked as something that causes the virtual memory system to issue a page fault. Concurrent JS will require a combination of inline caching and virtual memory tricks to make TID and SW checking cheap.

Inline caching means emitting different code for each property access, and then recompiling each property access potentially many times as we learn new information

about what this property access may do. Inline caches are able to get incredibly precise information about the behavior of each property access because we always emit a fast path access that can only handle exactly those cases that we have seen so far. We learn new information by recording everything we know about those accesses that failed the fast path. The complete log of failed accesses is then LUBed (least-upper-bounded) together to create a minimal set of `AccessCase`s. We can implement optimizations for new kinds of property accesses — such as the ones that have to check TID and SW bits — by considering in what ways a particular access site might be special and so specially optimizable. Below is a list of conditions we may encounter along with the strategy for how inline caches can test this condition and handle it.

- Probably many object accesses will always see TID = current and SW = false. These accesses can simply subtract `encodeButterflyHeader(currentTID, 0)` from the butterfly before accessing it. If the speculation was wrong, the virtual memory subsystem will issue a page fault because of non-zero high bits. We can catch this as a Mach exception or POSIX signal, and divert execution to the slow path. The subtraction of this constant can often be encoded directly into the subsequent butterfly access instruction. As a result, these kinds of accesses will not experience any overhead in concurrent JS versus what they have currently. Note that this optimization does not apply to transitions, which must install a new butterfly while atomically asserting that nothing bad has happened — we consider those further below. Note that this optimization slightly favors the main thread (and so all single-threaded programs) because if currentTID = 0 then we don't have to add anything.
- Probably many object accesses will always see TID = current and SW = true. These can be optimized identically to the previous case. This state is for objects that we already know can be written to from many threads, but that only happened after the last time that the current thread transitioned the object.

- Probably many object reads will see TID != notTTLTID and SW = anything. These just need to check that the butterfly's high bits are not notTTLTID. This can be done with a single compare/branch. Other than this, they can proceed exactly how they do in our current engine.
- Probably many object writes will see TID != notTTLTID and SW = true. This means that the object is being written to by many threads. We are also writing to it, but we don't need to set the SW bit because it is already set. This check can also be performed in a single compare/branch. Together with the read optimization above, this means that concurrent JS programs that share objects will typically only experience one extra cycle (for fused compare/branch) for butterfly accesses.
- Sometimes, an object write will see TID != notTTLTID and SW = false. This means having to set the SW bit using a DCAS, which sets the SW bit while asserting that the type header did not change. These accesses will be more expensive than their friends, but this only has to happen the first time that any thread writes to a shared object. Our inline caching infrastructure will probably make it so that writes that sometimes see SW = false and sometimes see SW = true (and don't necessarily see TID = current but never notTTLTID) will be branchy: they will have a fast path for SW = true and a slightly slower but still inline path for SW = false. In the next section, we will describe optimizations that require invoking functions on the structure when the SW bit of any objects of that structure gets set for the first time. Since this inline cache knows about the structure at the time of generation, it can ensure that the structure is already informed that objects of its type may have the SW bit set.
- Probably some object accesses will see TID = notTTLTID and SW = true. By convention, we'll say that it's not possible to have TID = notTTLTID and SW = false (internally in our VM, it's technically possible to transition an object without "writing" to it, but we'll pretend that those are writes anyway). These accesses can subtract `encodeButterflyHeader(notTTLTID, true)` from the butterfly before accessing it, and then they must perform an additional load when reading from

the butterfly. The extra load is necessary because of the segmented object model: the butterfly pointer points at a spine that we can use to find the fragment that contains the value we're interested in. This is somewhat more than one extra cycle of overhead, since it introduces a load-load dependency.

These optimizations leave an unresolved issue: transitions. Transitions now require acquiring and releasing a lock. We have to do this anytime we add any property. In the next section, we show how to solve this problem using *watchpoints*. Then we describe how to implement transitions that need to create a segmented butterfly out of a flat one.

Watchpoint Optimizations

Transitions are difficult to optimize. Every transition, including those that see TID = current, need to acquire the object's internal lock to ensure that they set the butterfly, adjust the object's type header, and store the new value of the property all in one atomic step. Luckily, we can dramatically improve the performance of transitions by using our engine's *structure watchpointing* infrastructure.

Each object has a *structure*. Many objects will share the same structure. Most inline cache optimizations begin with a check to see if the object's structure matches the inline cache's expectations. This means that when an inline cache is compiled, it has a pointer to the `Structure` object in hand. Each structure can have any number of *watchpoint sets* in it. A watchpoint set is simply a boolean field (starts *valid*, becomes *invalid*) and a set of *watchpoints*. When the set is fired (the field goes from *valid* to *invalid*), all watchpoints are invoked. We can add two watchpoint sets to `Structure`:

- `transitionThreadLocal`. This remains valid so long as all objects that have this structure have TID != notTTLTID.
- `writeThreadLocal`. This remains valid so long as all objects that have this structure have SW = false.

This enables the following optimizations on top of the inline cache optimizations described above:

- Transitions with TID = current and SW = false can proceed as they do in our existing engine so long as the structure's `transitionThreadLocal` and `writeThreadLocal` watchpoint sets are both still valid. This means not doing any extra locking or even CAS when transitioning the object. This works even if other threads are concurrently reading from the object. This works even when building up an object that eventually gets read and written to from other threads, since in that case the `writeThreadLocal` watchpoint set only fires on the structure that is the target of the final transition used to build the object. The last transition can proceed without locking because the watchpoint sets of the source of the transition are still valid.
- Any checks for TID != notTTLTID can be elided if the structure's `transitionThreadLocal` watchpoint set is still valid and a watchpoint is installed.
- Any checks for SW == false can be elided if the structure's `writeThreadLocal` watchpoint set is still valid and a watchpoint is installed.

The fact that we can elide TID != notTTLTID and SW == false checks means that reads and writes to those objects don't actually have to check anything; they just need to mask off the high bits of the butterfly.

Most importantly, this means that transitions that happen on the same thread that allocated the object don't need any locking so long as the structures involved have valid `transitionThreadLocal` and `writeThreadLocal` watchpoint sets. Structures are dynamically inferred by our engine so as to have a close relationship to the code that creates those objects. Hence, if you write a constructor that adds a bunch of fields to `this` but does not escape `this`, then the structures corresponding to the chain of transitions that occurred in the constructor will all have a valid `transitionThreadLocal` and `writeThreadLocal` watchpoint sets. To make sure that the object continues to have a flat butterfly, you need to either never dyamically add

properties to the object, or never write to the object on any thread other than the one that constructed it. Objects that play by these rules will experience almost no concurrency overhead, since property accesses will have at most one extra instruction (a mask) on the fast path.

Watchpoints, and the operations that cause them to fire, will execute under safepoint: if we ever perform some operation that finds that it has to invalidate a watchpoint, it will do the operation while all other threads are stopped as if for a garbage collection. This means that if some optimized code is performing non-atomic transitions involving some structures, and some other thread attempts to write or transition an object that uses any of those structures, then it will not actually perform the write until that optimized code reaches a safepoint and gets invalidated. Most of the time, the watchpoint sets will tell us that they have been invalidated even before the optimizing JIT compilers try to install any watchpoints. We expect few watchpoint invalidations in steady state.

To summarize, if our optimizer is able to guess which object properties you will add to an object at the time of allocation, then the cost model of your object accesses does not change at all, since inline properties get concurrency for free. If you do have out-of-line properties, then they will perform almost exactly as they do now (occasionally, an extra arithmetic instruction will be involved in computing the butterfly access) so long as either the object is only written to by the thread that created it (and read by anyone), or no new properties are added to the object after creation (in which case it can be read and written by anyone). If you break this pattern, then transitions will become about 7x more expensive and all other object accesses will become 2x more expensive. This slow-down will be extremely surgical: only those property accesses that touch segmented butterflies will experience the slow-down. This system is designed to let you dynamically and concurrently add stuff to your objects, and the hope is that if you do so in moderation, you won’t see much of a change in performance.

Arrays

Array element accesses will benefit from TTL similarly to the way that named property accesses do:

- Accesses to TTL arrays will be about as fast as they are now.
- Accesses to non-TTL arrays will require one extra indirection.

We handle array transitions a bit specially. Many array transitions in JavaScriptCore already use locking because of their need to avoid races with the garbage collector. Adding locking to the remaining array transitions may cause performance issues, so this section considers an alternative.

Resizing an array in response to an out-of-bounds store or something like `array.push` is the most common kind of array transition. Fortunately, this transition does not change anything in the type header of the object — it only changes the butterfly pointer. Whether or not the butterfly has array elements is reflected in the structure. Therefore, we can just use CAS on the butterfly pointer, and then have a rule that any changes to butterfly pointers of objects that have array elements requires CAS even if the object lock is already held. Because array structures' `transitionThreadLocal` and `writeThreadLocal` watchpoints are unlikely to be intact (any shared array use will invalidate them, since arrays share structures), we expect that even transitions on TTL arrays will have to use CAS in the common case. This butterfly pointer CAS is sufficient to ensure that a simultaneous attempt to make the array not TTL won't get confused by our resize operation.

One CAS for array resizing is probably cheap enough to be practical. The CAS is likely cheap relative to the current costs of resizing, which involves allocation, copying data, and initializing the newly allocated memory.

Objects that have both named properties and array elements will now have to use both locking and CAS on some of the transitions involving named properties.

Fortunately, objects that have both array elements and named properties are sufficiently uncommon that we can probably get away with increasing their named property transition cost a bit.

Quickly Transitioning From Flat To Segmented

Turning flat butterflies into segmented ones requires a special kind of transition. Fortunartely, this transition is cheap. Butterfly fragments contain only data. They look just like fragments (fixed-size slices) of the payload of a flat butterfly. Therefore, we can transition a flat butterfly to a segmented one by allocating a spine and pointing its fragment pointers at the original flat butterfly.

Any read or write to the flat butterfly while the butterfly is being transitioned to segmented will seem to users of the segmented butterfly to have happened to the fragments. In other words, although some threads may mistakenly think that the butterfly is still flat, their accesses to that butterfly will still be sound even after the butterfly is already segmented.

Making this work right requires making sure that the segmented butterfly's fragments share the same memory layout as the flat butterfly that they were converted from. For this reason, the first array fragment contains the public length and the *old* vector length; it will remember the vector length that the flat butterfly used forever, and the real vector length will be in the segmented butterfly's spine. This ensures that if there is a race between a flat butterfly array access and a flat-to-segmented transition, then the flat butterfly access will correctly know the flat butterfly's size because its vector length will not change. To line up the object models, we also store out-of-line properties in reverse order in the out-of-line fragments, to match what flat butterflies do.

This works so long as the flat butterfly accesses loaded their butterfly pointer before the transition occured. If they load it later, then their TID check will fail, probably in the form of a page fault on an access to the butterfly.

Deletion and Dictionary Lookups

JavaScriptCore tries to make objects share structures whenever possible. This relies on two properties of structures:

- Structures are immutable.
- When a new structure is needed, we generally hash cons it.

This is a great optimization if it indeed causes objects to reuse structures. But some objects will have a unique structure no matter what the VM tries to do about it. JavaScriptCore tries to detect when this happens, and puts the object in *dictionary* mode. In dictionary mode, the structure has a 1:1 mapping to the object. Also, the structure becomes mutable. Adding and removing properties means editing the structure and the object in tandem.

Deletion is closely related to dictionary mode, since deletion will immediately put the object in dictionary mode.

It's already the case that mutations to a dictionary require holding the structure's lock. This is necessary to support concurrent JIT and concurrent GC.

To support concurrent JS, we only need to make these changes:

1. Reading from a dictionary requires holding the structure's lock, in case some other thread is changing the dictionary.
2. Properties added before the object entered dictionary mode must be treated specially by deletion.

We are not worried about the performance hit of grabbing a lock for all reads of a dictionary. It's already relatively expensive to read a dictionary.

The existing facilities for safely transitioning objects will naturally support transitioning to dictionary mode. Usually, dictionary transitions don't involve deleting properties. They

happen when we detect that the program is adding so many properties to the object that it's probably going to perform better as a dictionary. In that case, the fact that some other thread may be in the middle of accessing this object without holding any locks does not matter. For non-dictionary objects we require that only transitions take the lock. Reads and writes involve separate steps for checking the object's type, loading the butterfly, and then accessing the butterfly. But if none of the properties that were added before the dictionary transition get deleted, then it's fine for some other thread to race when accessing those old properties. We will call this the phenomenon of tardy accesses. Even though those *tardy* accesses don't do any locking, they are correct for the same reasons that they are correct before the object becomes a dictionary. That problem is up to the butterfly object model, which will be either flat or segmented depending on whether the object is still TTL. Dictionary mode operates orthogonally to TTL inference and butterflies.

But if any of the properties added before the dictionary transition get deleted, then we have to handle this deletion specially. Normally, deletions cause the deleted slots to become reusable. We cannot do this here, because then a tardy read to some deleted property `f` might end up reading the value of some newly added property `g`. Or, a tardy write to some deleted property `f` might end up overwriting the value of some newly added property `g`. We can prevent either of these bad outcomes by simply not reusing the space freed up by deleted properties, if those properties had been added before the dictionary transition.

This does not lead to unbounded memory usage. When the GC does its safepoint, it already knows that all memory accesses have completed. Therefore, the simplest implementation is to have GC change the state of those deleted properties when it visits the object. Once the GC flags those property slots during a safepoint, future property additions can reuse those slots.

Inline Caches

Once we enable concurrency, recompiling an inline cache becomes harder because you need to be careful when changing code that may be currently executing on another CPU. We plan to have a tiered defense against this problem: we will infer the thread-locality of code to use the same inline caches as we do now as much as possible, we will buffer inline cache changes whenever it's safe to do so, and finally we will rely on safepoints if an inline cache needs to be reset eagerly.

We expect that in many programs, inline caches will reach steady state before the code ever executes on more than one thread. Hence, we plan to have each block of code track its thread-locality. So long as it has only ever executed on one thread, it will remember this and check for it on entry. So long as this is true, any inline caches within that code can be modified without any extra synchronization. We can track thread-locality on as fine-grain a level as we like; for example it can even be per-basic-block. It's probably most natural if we use JavaScriptCore's notion of `CodeBlock` as the granularity; this roughly corresponds to functions in the source code.

Once the thread-locality of code is broken, we can switch to buffering inline cache changes. Our `PolymorphicAccess` inline cache infrastructure already buffers changes because that's most efficient even if we don't have to do expensive synchronization. For inline caches that may be executing globally, we can globally buffer all changes. For example, the VM can perform once-per-millisecond safepoints to flush changes to all global inline caches.

Sometimes, we need to make an immediate change to an inline cache. When this happens, we will rely on our ability to safepoint the system — i.e. to stop all threads at a point where we can account for each thread's state. This is not a fast operation when there are many threads. Java virtual machines already have to do something like this for class hierarchy invalidation and biased locking. By design, these eager invalidations are unlikely. We only perform optimizations that can lead to eager invalidations if we have profiling data that suggests that it would be unlikely.

Local Optimizer Lock

So far, this algorithm relies on being able to sneak 16 bits of information into the high bits of the butterfly pointer. Some hardware will not allow us to do this. For example, fewer than 16 high pointer bits may be used for the all-zero check in virtual memory. If the system only allows for 8 checked high bits, then we will only be able to support 127 threads. Concurrent JavaScript would probably still be useful even if it had a hard 127 thread limit, but this would be an unusually low limit to impose at the language level. This section shows how to overcome this limit.

If the object model can only handle thread-locality inference for 127 threads, then we can either choose to have the 127th thread get no thread-locality inference or we can try to map more than one logical thread onto the same thread identifier. Threads mapped to the same TID won't be able to execute concurrently to each other. To do this, we can borrow the idea of the GIL (global interpreter lock) from Python. The CPython implementation only allows 1 native thread to interpreting Python code at a time. It achieves this by having a lock (the so-called GIL) that protects the interpreter. The lock is released and reacquired periodically, which gives the appearance of concurrency. Since we can also release the lock around any potentially-blocking operations, we can even avoid deadlocks that arise from insufficient concurrency. We can apply this technique here: if we are limited to 127 threads, then we can have 127 locks protecting JS execution. So long as there are 127 or fewer threads, this lock will not do anything; anytime we try to release it, we will do nothing because the lock will tell us that nobody is waiting to acquire it. "Releasing and reacquiring" the lock will really be a cheap load and branch to verify that there is no need to release it.

This lock will be local to a thread pool rather than global to the whole engine, and it will protect all of our optimized paths rather than just protecting the interpreter. Hence the name: local optimizer lock, or LOL for short.

Thread-Unsafe Objects

DOM objects behave like JavaScript objects, but are actually a proxy for complicated logic implemented in C++. That logic is usually not thread-safe. The native code that supports the DOM transitively uses many native APIs that are meant to only be used from the main thread. It would take a lot of work to make the DOM completely thread-safe. Our strawman proposal only requires making the DOM global object capable of handling lookups to self properties, which we need to allow threads to access global JavaScript properties like `Object` and `Thread`.

In WebKit, variable resolution and self property resolution on the `JSDOMWindow` largely follows a pattern of relying on existing JS property lookup mechanisms. It uses exotic behavior when the window has a null `frame`. We already support installing a watchpoint on the non-nullness of `frame`. Hence, we can support fast concurrent accesses to properties of `JSDOMWindow` while honoring the `frame` special case by having threads use the existing watchpoint set. This implies that some of the `JSDOMWindow` slow paths will need locking, but that's probably acceptable since the majority of global object accesses are inline cached.

Native apps that want to dip their toes into concurrent JS may also want to restrict sharing for some of their classes. The thread-affinity check on accesses to objects will need to be implemented in the VM itself to give our compilers the ability to optimize it away. This means that it's possible to expose the functionality to C/Objective-C API clients.

Summary

We think that we can make it possible to execute JavaScript concurrently in WebKit while providing good enough performance that developers would enjoy using this feature. At the heart of this technique is the combination of segmented butterflies, transition thread locality inference, and lots of cheap per-object locks. So long as programs obey certain behaviors, they should experience minuscule overheads: property accesses will cost about as much as they do today and objects will not require any extra memory.

If some objects decide that they don't want to play by our rules, they will get slightly higher overheads.

Related Work

Segmented butterflies are a straightforward combination of the array object model in the Schism garbage collector and the butterfly object model that we have used for a long time in JavaScriptCore.

Transition-thread-locality inference is inspired by work on biased locking. As in the HotSpot biased locking scheme, we use types to reason about whether we have a guarantee of thread locality. Unlike that scheme, we don't rely on deoptimization to break a thread-object relationship. Our primary mechanism of informing other threads that an object is no longer transition-thread-local is to have different tag bits in the butterfly pointer.

The combination of segmented butterflies, transition-thread-locality inference, and the ability of both schemes to fall back on per-object locking when things get weird is not something that we have seen before. Most object-oriented systems that allow concurrency do it either by having an object model that naturally avoids expensive-to-resolve races, like in Java, or using some mechanisms to restrict the amount of concurrency in the language.

For example, CPython uses a global interpreter lock (GIL) to ensure that the interpreter never races with itself. This is a powerful technique that captures some of what concurrency is about: once I/O or other kinds of blocking happen, you want your other threads to be able to do some useful work in the meantime. The GIL allows this, so it makes threads useful to a large class of applications. Unfortunately, it doesn't allow programs to exploit parallelism. The best feature of the GIL is that it is easy to implement. For example, it could be used to implement our proposed thread semantics in any JS engine. The scheme in this post is all about full concurrency optimized for 64-bit platforms.

An effort called the *Gilectomy* is underway to remove CPython's GIL and replace it with fine-grained locking and clever atomic algorithms. The Gilectomy is not yet as fast as stock CPython; both single-thread and multi-thread programs run significantly slower in the Gilectomy. The performance problems appear to be related to locking in the allocator (we plan to use thread-local allocation buffers) and in the reference counter (we don't have reference counting). The Gilectomy does not remove locking from accesses to objects. Like JavaScript's objects, Python's objects are dictionaries that can resize dynamically. Most of our scheme is about how to make accesses to those objects fast even when multiple threads are reading and writing to the same object.

PyPy also has a GIL removal effort underway, but they haven't said much about how they plan to handle synchronizing object accesses aside from using locks. We will also have locks, but we also came up with optimizations to avoid locking in most cases.

Another approach was the *title locking* scheme in SpiderMonkey. It has since been removed. That scheme also involved per-object locks, but did not have our optimizations for avoiding locking in most cases.

Daloze, Marr, Bonetta, and Mossenbock propose a scheme for concurrency in Truffle, a runtime that supports many dynamic languages including JavaScript. That scheme requires objects that are shared to use synchronization for all writes. Sharing is detected by a write barrier. When a thread-local object is stored into a shared object, the runtime will traverse objects transitively reachable from the escaping object. All of those objects are marked shared so that future writes to those objects use synchronization. Our scheme does not require synchronization on shared writes, except when those writes are transitions. Our scheme does not require traversing object graphs; transition thread locality inference is on a per-object basis. A TTL object may point to a non-TTL object or vice-versa.

Cohen, Tal, and Petrank recently introduced the *layout lock*, which allows fast reads and writes to objects that may have their layout changed. Readers need an extra load (which has to be fenced or dependent), writers have to acquire a read lock (which can be as cheap as a pair of fenced loads and a branch), and transitions have to acquire a write lock and do some extra book-keeping. We suspect that this has a similar cost model to segmented butterflies, except for writes. Segmented butterflies only require an extra dependent load for writes, which is a bit cheaper than acquiring a read lock, which requires two loads fenced around a store. Fencing loads around a store is inconvenient; for example forcing a store to happen before a load on x86 requires an `xchg` instruction for the store, which is more expensive than just doing a `mov`. On ARM it requires using acq/rel variants for the loads and stores. On the other hand, the extra dependent load for writing to a segmented butterfly requires no extra fencing on either x86 or ARM. Transitions and reads are probably equally fast with segmented butterflies and layout locks, but writes matter a lot to us. Writes happen often enough that segmented butterflies are likely to be noticeably faster than layout locks for the JavaScript object model. On the other hand, the method of segmentation is not easy to reproduce in many different kinds of data structures. Since it is more general, the layout lock could be useful for more complex data JSC data structures like `SparseArrayValueMap` and `Map` / `WeakMap` / `Set`. Those data structures may be too complicated for segmentation, but not too complicated for the layout lock. Like segmented butterflies, layout lock can be combined with transition thread locality inference to avoid doing any of the locking until it's needed to protect racy transitions.

Conclusion

This post shows how WebKit's JavaScript implementation can be modified to support threads. Our scheme allows for all JavaScript objects to be shared. Our scheme avoids locking on fast paths, like the fast paths for property accesses. We expect it to be possible to implement this scheme with low overheads. Serial code will experience no overhead, and concurrent code will only experience big

overheads if it adds properties to an object that is being written to by multiple threads.

The next step for this wild thought experiment is to attempt to implement it and see if it works. We will be tracking progress under bug 174276.

Update: The original version of this post did not cite Daloze et al's OOPSLA '16 paper on concurrency support in Truffle. We have updated the post to include a comparison between our scheme and theirs.

Next

Release Notes for Safari Technology Preview 39

Learn more

Previously

Release Notes for Safari Technology Preview 38

Learn more