Shared-to-unshared references for Shared Objects in V8

Attention: Shared Google-externally

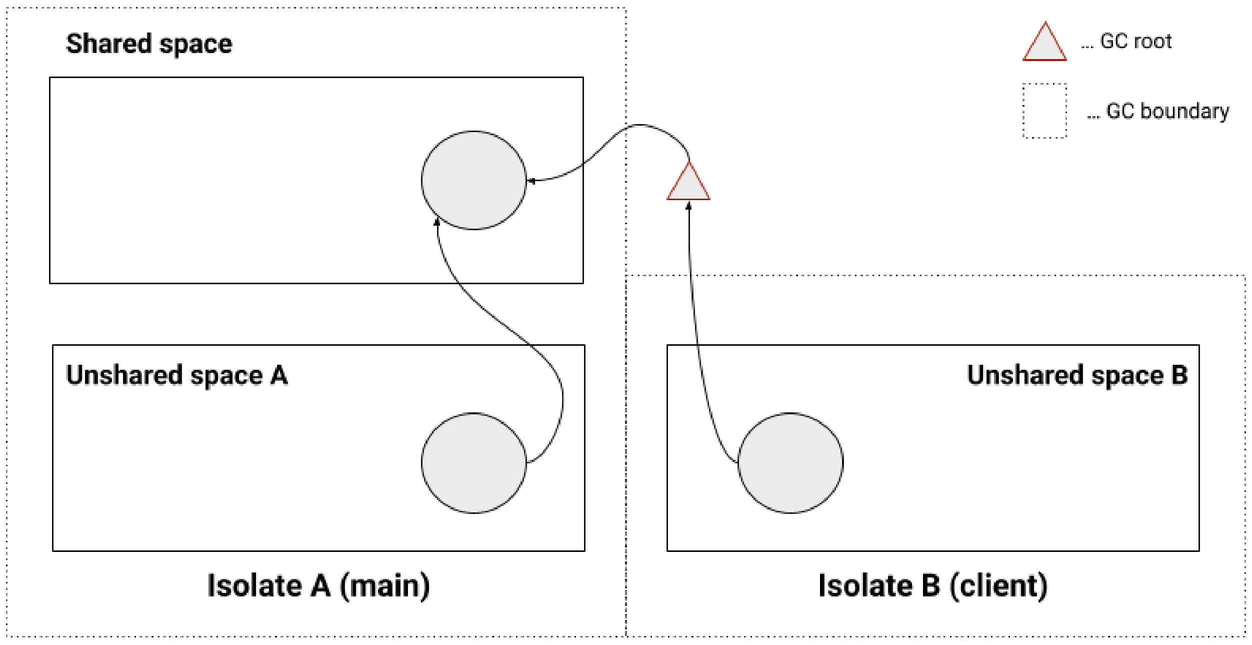
Author: Michael Lippautz , Jul 24, 2024
Contributors: Shu-yu Guo

The doc provides an overview of the problem of shared-to-unshared references in the context of shared V8 objects.

Background and motivation

A *shared space*, which was initially implemented via a single separate V8 isolate that was shared, is a way to create shared V8 objects for JavaScript (JS) and WebAssembly (Wasm). In V8, a shared space is implemented as a single shared old generation space that is owned by the first isolate – also called *main isolate* – that is created. All subsequent isolates created are called *client isolates*.

Memory in the shared space is currently only reclaimed in full garbage collections (GCs) on the main isolate. References from unshared to shared objects are modeled as regular references for the main isolate and roots for the client isolates.



It's trivial to see that this design can leak memory when references are held from client isolates as those are always treated as roots.

When this design was initially proposed it seemed the simplest way to get a shared space up and running while preserving local garbage collections for each V8 isolate. The big caveat was that it can leak some memory (due to maintaining roots) and doesn't actually support back references (either explicit or implicit) from shared to unshared objects. See the discussions in Shared JS Objects and Shared heap implementation. Already back then it seemed likely that even JavaScript may need back references or a primitive that emulates them from shared to unshared objects eventually.

With Wasm adding GCed types in WasmGC and its multi threaded version (being specified currently) there's a need for modeling Wasm objects as always being shared while at the same time requiring primitives to connect such objects back to JavaScript via web APIs and allow interacting with the DOM. Ultimately, this results in the need for shared-to-unshared references which can either be modeled via thread-local storage (TLS) memory, or in a simplified model via thread-bound storage (TBS) which is always checked at runtime. Note that this doesn't necessarily imply direct references but merely requires a high-level primitive that acts as such a reference.

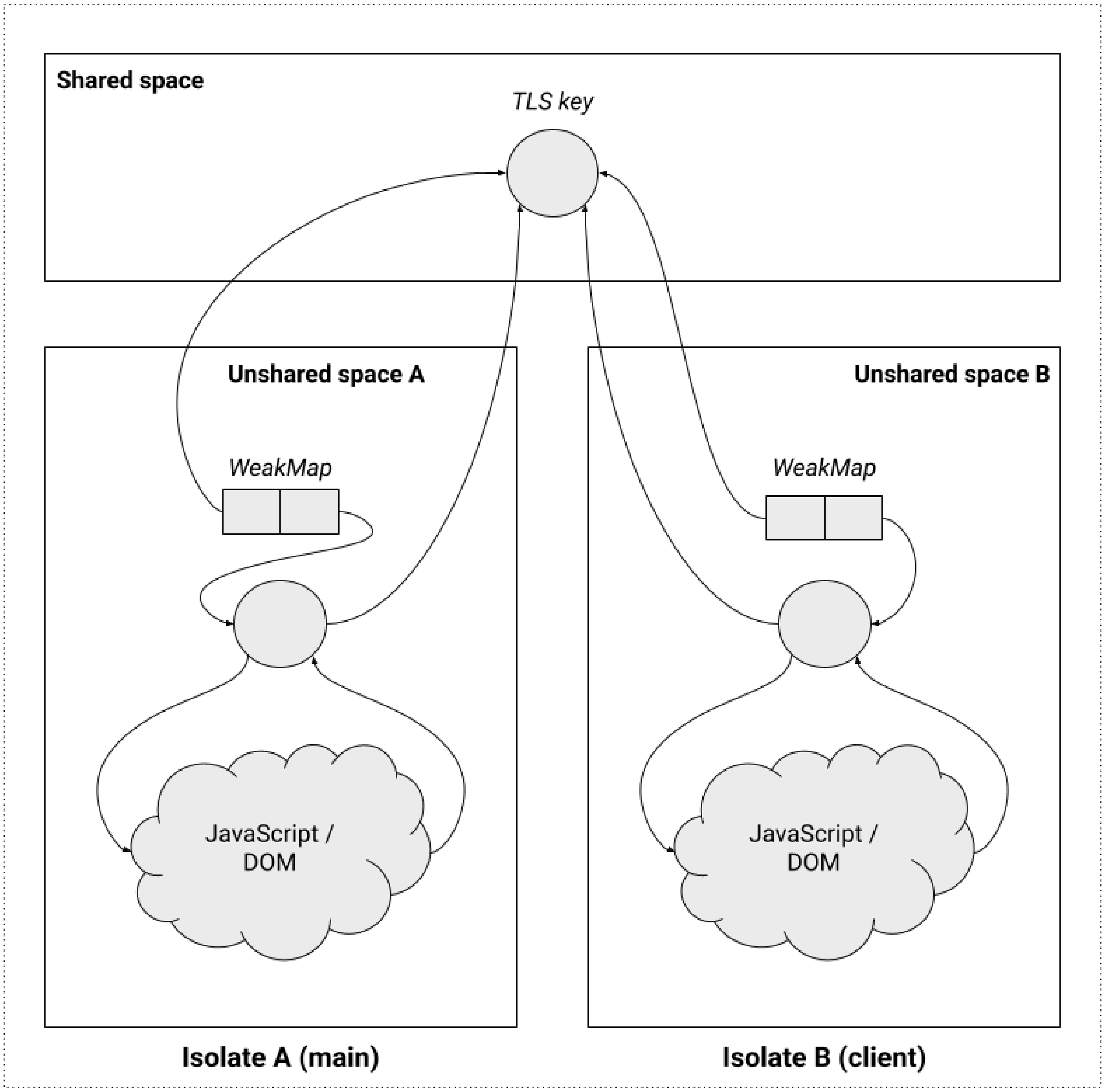
While dynamic TLS via e.g. Java `ThreadLocal<T>` may be a rare use case, connecting WasmGC objects back to the DOM seems very much required for binding multi-threaded code back to the DOM (as the web's UI framework). One can argue that main thread TBS is what's actually necessary here and TLS would merely be a generic primitive to support this case.

The rest of the document will explore options for TLS and TBS primitives from a V8 perspective. It mostly ignores other terminologies and specifications and merely models scenarios using object graphs using primitives like JavaScript `Map` and `WeakMap`.

Land of milk and honey: A global garbage collected heap

In a design with a global garbage collected heap a group of isolates would be managed by a single garbage collector. In such a design, a shared space would be accompanied by local spaces. Traditional JavaScript and the DOM would live on a local space whereas WasmGC multi-threaded objects would just be allocated in a shared space. References between all shared and unshared objects in all directions would be possible.

TLS/TBS in such a design could be implemented via JavaScript `WeakMap` where the key is a shared object and values are local objects.



Advantages:

- Cycles are fully collectable.
- No need for explicitly finalizing TLS entries or clearing references.

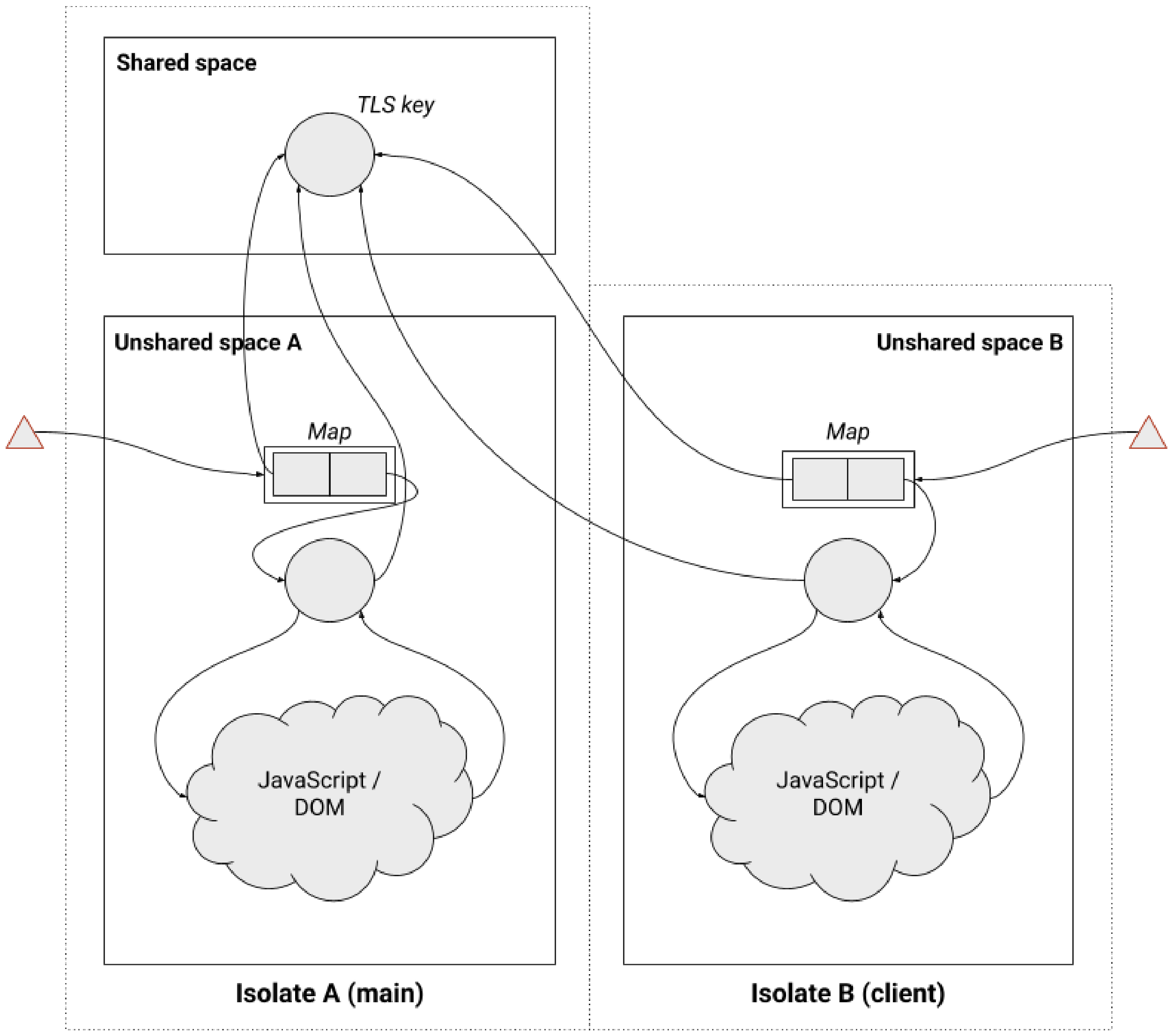
- The design would allow for fine-grained configuration. We could add *isolate groups* for isolates that share a global heap and those could exist multiple times in a single process. If starting from scratch, this is likely what one would implement as it allows full flexibility.

Disadvantages:

- V8/Blink implementation does require global refactorings and changes. E.g., WebWorkers must be wired up into a global heap.
    - Oilpan could stay local in general, even though they already contain cross-thread references. In a sense such references would become regular traced references rather than roots into other heaps. While this would be a better design, there's some issues to be expected in this area.
- The biggest problem with a fully global heap is the need for a global safepoint. All of the web platform implementation code is just regular C++ code though, sometimes even calling out to third-party libraries. None of this is managed runtime code that would provide safepoint interrupts. As a consequence, it's unclear what the time-to-safepoint in a fully global heap would be.
    - Oilpan had safepoints but they were removed because of perf problems (doc).

A fully explicit approach

In a world without garbage collection support for reclaiming cycles a model could use tables represented as JavaScript `Map`. We would allow keys to be shared objects and values to be unshared objects. Keys would always root the shared object.



Since the table itself is a local root, this design is equivalent to something that avoids explicit references by merely using indices. The design is equivalent to this proposal.[1]

In such a design cycles would by design be leaky unless explicitly broken via clearing references. New APIs must be specified to actually break cycles and enable TLS/TBS use cases without leaking memory.

Advantages:

- Very simple and implementable with minimal effort on current GC infrastructure.
- No need for global safepoints. GCs are still fully independent.

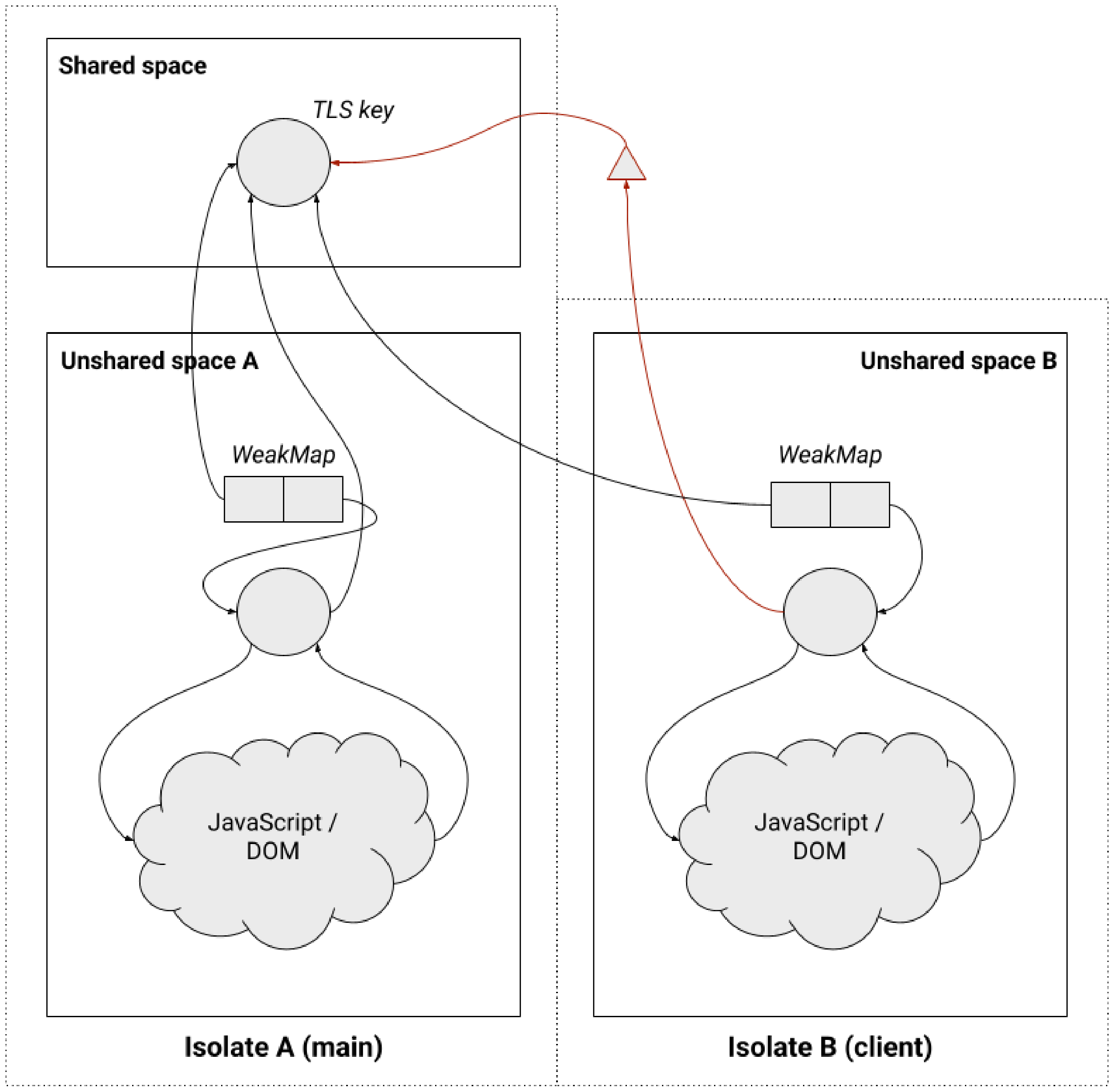
[1] https://github.com/WebAssembly/shared-everything-threads/issues/75

Disadvantages:

- Doesn't collect any memory automatically and requires explicit clearing. Even with explicit clearing it is not obvious whether certain memory can be collected as it's not clear whether applications can actually reason about internal object graph edges.

WeakMap as a way to implement main-thread TBS

This design explores a version of using `WeakMap` in V8's current design where references from client isolates are kept as roots and the shared space is only reclaimed from the main isolate.



This design can be allowed in V8 by merely allowing shared objects (e.g. TLS keys) in regular `WeakMap`s. It's trivial to see that this architecture will leak memory and requires explicit clearing as soon as there exist back references into the shared space from client isolates. However, cycles on the main isolate are fully collectable.

An extension to this design could allow switching ownership of the shared space at the cost of prohibiting interleaved GCs. The isolate that owns the shared space would be allowed to treat its own references as regular references whereas all other isolates' references would be treated as roots. All remembered sets would need to be recorded which includes the main isolate. This would allow reclaiming TBS references for all threads that are used exclusively. Switching could be opportunistic as in the worst case scenario we can always run a local GC with treating unshared-to-shared references as roots.

Advantages:
- Same worst case scenario as the fully explicit approach for TLS.
- Allows reclaiming TBS on the main isolate which on the web is the isolate with actual DOM access. In that sense it's a strict improvement over using a strong `Map`.
  - Allows for reclaiming all of TBS with the extension of switching space ownership.
- No need for global safepoints. GCs are still independent.

Disadvantages:
- Not all cycles are reclaimable. Specifically, TLS (as in shared objects used as keys in multiple local `WeakMap`s) still form unreclaimable graphs.

Distributed global heap

Building on top of the previous design, we explore what's necessary to reclaim cycles in general and how to adopt a design for V8 that does not require implementing a fully global heap while at the same time allowing incrementally migrating towards it.

The current design in V8 that distinguishes main and client isolates suffers from the fact that it introduces root references which may take part in cycles that require manually breaking. While that does hinder collection of cycles, it does enable local garbage collection which is beneficial to avoid global safepoints.

With a design based on `WeakMap` we can implement a mostly incremental and concurrent global GC for V8 in the following way:
- Heaps are generally local and garbage collections can be local as well.
  - This requires maintaining the remembered set from unshared to shared heaps which V8 already has.
- A full global GC requires all local GCs to participate.
  - In such global GCs, the individual local GCs may start independently!
- The shared space would still be collected by the main isolate. This allows for TBS on the main thread to just work without any extra overhead (see below).

The GC then works as follows[2]:

1. Main isolate starts a global GC.
2. Client isolates start their GC afterwards. All GCs are wired up to have `Worklist` references to communicate references; this is very similar to the unified heap setup.
3. Any references from shared to unshared are added to the corresponding local GC's `Worklist`.
4. Upon reaching the end of marking on the main Isolate, a global safepoint is requested.
5. Such a global safepoint may have a timeout of X ms. Upon reaching that timeout a main isolate may decide to run as a local GC, considering the remembered set of client Isolate references as roots.
    - This decision can be dynamic per isolate. Whichever isolate enters a safepoint on time can participate in the global heap. All other isolates are treated as local.
6. In case all clients reach the global safepoint in time, we continue marking on all local GCs, communicating unshared to shared references via V8 `Worklist` mechanisms.
    - Note that safepoint here means "willing to participate". This may be different from "parked" state. In the worst case parked would mean not processing this heap and relying on roots.
    - This can use parallel marking as we have today.
7. Eventually, a fixed point will be reached in the marking phase upon which the GCs can diverge again. I.e., the local GCs can independently sweep their heaps.

Advantages:

- Mostly-local GCs are still possible.
- No large API refactorings required. E.g., WebWorkers would use a different isolate and local heap as it does today.
- Global GCs have bailouts for when a global safepoint is not reached on time.
- It's possible to add this incrementally on top of a design using `WeakMap` that would just leak.

Disadvantages:

- Complexity is non-negligible.

[2] Arguably, Shu-yu Guo already prototyped something "similar" that instead of computing the full transitive closure would use marking state snapshot (on the client isolate) and avoid delegating to the different GC here.

Assuming a VM that cannot provide incremental and concurrent GC and doesn't have ways (or want to) communicate references globally, there's also an alternative design for stop-the-world GC:

- A full global GC requires all local GCs to participate.
- All TLS keys receive an extra bit per participating isolate indicating lack of liveness from their own transitive closure. Let's call this the *unreachable TLS bit*.
    - The number of bits is dynamic in the participating isolates and requires hooks for entering/leaving the system.
- TLS keys are known and can be iterated.

The GC then works as follows:

1. All client isolates start local GCs that include shared space.
    - Let's call this the *initial phase*.
    - The GCs do not consider references from other isolates as roots in this initial phase.
    - The GCs consider references from their own heaps into the shared space as regular traced references (not roots!)
    - The GCs are performed sequentially.
2. Each local GC then iterates all TLS keys that are unmarked to set the *unreachable TLS bit*.
3. Each local GC then considered references from other isolates as roots and finishes marking the transitive closure.
4. At the end of each local GC the shared space is unmarked (as it's not actually owned by a client isolate).
5. After the initial phase, all TLS keys that have all their unreachable TLS bits set are candidates for reclamation.
6. The main isolate then performs a GC the same way as the client isolates
    - The difference is that it can actually reclaim TLS keys that have all their *unreachable TLS bits* set before considering the roots from other isolates.

Because any state can change in-between sequential local GC runs and effectively create a new path to a TLS key all of the GCs must happen in one stop-the-world pause. The algorithm will get rid of TLS keys and is thus able to reclaim cycles. It requires multiple GCs to eventually clean up memory as initially only TLS keys will be removed from shared space. The whole procedure is expensive but in theory allows reclamation of unreachable islands containing shared-to-unshared references.