We propose the existence and functionality of a special shard that we call Master Shard

(MS). The main intent is for it to act as cache for frequently-used contracts and (frequently-used & rarely-modified) data from all the normal shards and bring them into the “global scope”

, optimizing thus the inter-shard operations.

The Beacon Chain, along with the MS will always be synced in a node. Calls to MS will be faster and cheaper than calls to other shards.

One option was to keep the Beacon Chain EEs stateless. Then, the MS could also hold the last state roots for all shards. This is similar in concept with a master database, where databases, tables, and fields are kept as records.

Master Shard Transactions

The MS only has three allowed methods:

- Add

, returns the

corresponding to the stored resource

- Update
- Get

, returns the cached

can be a contract or an instantiated value type - e.g. uintX

, boolean

, array, struct

.

Writing to the Master Shard

Writing to the MS cannot be done through a normal, user-initiated transaction. This is a process controlled by the Beacon Chain’s EEs.

Block Producers (with the help of Relayers) and Validators run the EE scripts on the shard block data, in order to get the post-state transition shard root hash. This will get sent to the Beacon chain by the Validators.

A simplified EE can be viewed as a reducer

function: from the previous shard state and current transition to the post-transition shard state.

function reducer(shard_prev_state_root, block_data) { // reduce, hash etc. return shard_next_state_root; }

The output can also contain transaction receipts or other by-products returned by executing the transactions inside the VM (based on eWasm in Eth2).

The VM can determine if a resource (e.g. smart contract) from Shard1 has been frequently used by other shards and decide to move it to the Master Shard. It will build the necessary state transitions for this, run them and get the receipts.

Our simplified EE example becomes:

function reducer(shard_prev_state_root, ms_prev_state_root, block_data) { // get MS state transitions from block_data and add them to the previous MS state // reduce, hash etc. return [shard_next_state_root, ms_next_state_root]; }

The new MS state transitions, along with the previous MS state root hash, will go through the same process of being reduced. The new output will contain both the final state root hashes: for the initial shard and for the MS. Additionally, it will also hold all the transaction receipts.

Writing to the MS at this stage, should not cost the transaction initiator more. Gas estimations are still an open subject.

High-level sequences of what happens before and after adding a contract to the MS:

[

d6cda817-4262-4253-bb3b-0e9cd8726934

908×546 19.9 KB

](https://ethresear.ch/uploads/default/original/2X/0/0a6e05d560fac1ab18c085d4551df8e18bcaa91d.png)

[

8e697fe8-8f86-4ab6-94eb-a386f658b092

722×384 13.4 KB

](https://ethresear.ch/uploads/default/original/2X/6/654f215a136b325eb6805934f6d80af1caff11cc.png)

When Is a Resource moved To the MS?

I mentioned that the VM looks at how frequently a Shard1 resource is used and determines if it should move it to the Master Shard.

This requires a counter for how many times the resource is used by other shards (reads or writes): cache_threshold

- a global variable (per resource), with the maximum number of uses before the resource is cached.

The cache_threshold

should be modifiable in time - if the number of total Ethereum transactions increases substantially, the cache_threshold

might need to be higher.

The resource counter

can be stored on the MS itself, in a smart contract. It can be a key-value store, where the key

is the resource address, which already contains the shard identifier and the value is the current counter value. If it is possible to only cache smart contract storage partially (per data record), then the key

might also contain the data record identifier.

The counter is increased by the EE every time the resource is read or written. This will happen with any cross-shard transaction. Gas costs remain unspecified, but it should not be expensive, as this should be deterministic and part of the system (not initiated by the user).

Updating Resources on the Master Shard

If a transaction is sent to a smart contract on Shard1, triggering a change of a resource that is cached on the MS, the EE will also update the cached resource, using the Update

MS transaction.

Removing Resources from the Master Shard

The easiest solution would be resetting the MS after a number of blocks (e.g. equivalent of 1 year), removing the cached resources.

There are other solutions, but more complex or computationally intensive, that we will propose.

Reading from the Master Shard

Each time a transaction sent to a shard requires to read data from another shard, it will first look in the global scope (MS), to see if the data is cached. If it is not, the transaction will continue normally. If it is, it will use the global scope. The MS enables data memoization.

Conclusions

- avoids redeployment of libraries and contracts to multiple shards
- cross-shard transactions will be faster if read data is cached

- there are gas costs to caching data which should be clarified in more detail