We know that with merge blocks and clever use of fork choice rules, we can have a sharded chain where a block on one shard is atomic with a block on another shard; that is, either both blocks are part of the final chain or neither are, and within that block there can be transactions that make synchronous calls between the two shards (which are useful for, among other things, solving train-and-hotel problems). However, there remains a challenge: how do we do state execution for such blocks? That is, suppose that we have block A on shard 1 and block B on shard 2, which are merged; how does a node which is a full client on shard 1 but a light client on shard 2 execute the merged blocks, when the execution clearly depends on the pre-state of shard 1 and shard 2.

One possibility is if we operate in the stateless client model, in which case any nodes that need to execute that block could simply download the witnesses from all blocks, and execute it. In general, it's not reasonable to expect clients to have the full state of all shards, so if we generally expect clients to hold the full state on one shard, then they will be stateless on the other shard. We can expect merge transactions to hop between the two shards many times; for example:

x1 = value_on_shard_1 x2 = sha3(x1, value_on_shard_2) x3 = sha3(x2, another_value_on_shard_1) x4 = sha3(x3, another_value_on_shard_2) .....

This precludes the possibility of clients that are full on one shard and light

(ie. non-executing) on the other shard, as one cannot complete the execution without seeing all of the intermediate steps, and it is impractical to pause execution many times to download light client proofs for all the intermediate steps. This implies that a setup is required where all transactions in the merge block come with access lists and witnesses (as in the stateless client spec), so that clients who just see the state root for both shards can execute it.

Note that even with this setup, the specific witnesses required can only be calculated after the post-state of the direct parent of the merge block has been computed. This implies a round of network latency between computing the state at block N and computing the state at block N+1, which is an efficiency limitation.

It is also worth noting that merged state execution technically does not require merge blocks. Instead, one can simply define a deterministic function that assigns shard IDs into pairs at each block number (eg. if we want to cycle evenly, and there are k

shards and k

is prime, then at block number N shards, shuffle shard IDs with i -> (i * N) % k

and then merge-execute blocks 1 and 2, 3 and 4, etc using post-shuffling IDs). Merge-executing two blocks would simply mean that a transaction starting on one block is capable of reading and writing state on the other block.

Another possible construction is to cycle through which shard is the "primary" shard during each block height, and during that height allow transactions from that shard to read and write to all

shards. All executors of all shards would statelessly process the transactions from that shard first, then process their own transactions.

The reason why merge blocks are not required is that state execution can be separated from consensus on data, and so if absolutely needed state executors can simply wait until blocks on all shards are finalized up to some height before calculating state up to that height (though a more optimistic approach, using conditional claims, is also possible).