@vbuterin's recent Phase 2 Proposal 2 document presents an "execution environment" concept: contracts on the beacon chain that hold independent per-shard states. Shards run their various blocks in these environments, updating their personal state, which is eventually committed to the beacon chain, presumably when the block is attested or via a crosslink (I can't find a definite statement on this, but I think the details don't matter in this post, so long as the states eventually reach the beacon chain).

My point in this post, in brief, is that this introduces a new perspective on the nature of shards, and that this perspective admits an expansion to a more nuanced shard structure capable of cross-shard communication. This expansion is compatible with everything that has already been done for sharding, but also adds a huge capability to shard-level metaprogramming.

Right now (as of Proposal 2), the beacon state tracks a list of execution environments, each of which has some code and also tracks a list of shard states corresponding to calls made to that environment in each shard. I like to "commute" this description: as a pure function, an EE is essentially an export from some notional "library", which may be instantiated into an actual smart contract by giving it a mutable state object; therefore, the EE list is more conceptually understood as being a data store of contracts, each of which is called by exactly one shard at various times (a single shard may call multiple contracts in its lifetime).

This creates a picture of the beacon chain as a set of noninteracting streams of contract calls, and each stream is part of a shard. Though this resembles a set of separate, parallel execution threads (i.e. many independent Ethereum-1.0s), by existing together in one central beacon chain what we actually have is a single smart contract universe in which only transactions may make calls, but not contracts themselves. The activity of sharding is then to parcel out maintenance of the contracts to separate consensus environments.

In this model, cross-shard communication would simply be the ability of one transaction to call multiple contracts. This means that transactions would not easily be divided among separate shards, but for execution purposes, it doesn't matter: clients can still have a set of EEs (contracts) they want to watch, and choose to process only the transactions that call them. This is the same kind of behavior that clients of the current sharding model would exhibit.

This gives a kind of static interaction among contracts ("shards"): transaction messages would be able to write programs that call contracts, call pure functions on the results, and call other contracts with those

results, so long as no jumps are allowed, so that we can still identify which contracts are affected. A form of conditional logic could even be included if the return values are augmented with a Nothing

or other exceptional value that would propagate through subsequent contract calls on that value. This is more than sufficient for many useful purposes, the most common of which would surely be to pay contract A with money from contract B. This logic, programmatic though it is, would not even require gas, since the length of the program and the duration of its execution are essentially the same.

What about consensus? Part of the point of sharding as it stands is to distribute the actual activity of running a blockchain, regardless of what it carries. I think this could still be done. Two observations:

1. The existence of validators who run transactions is necessary in order for consensus to include an affirmation of state updates.

2. Absent concerns about state transition, there is no reason that all transactions couldn't just be lumped into a single blockchain without validation (i.e. "data availability").

So let's take a two-stage approach to sharded consensus.

- Stage 1:

Blocks are created indiscriminately from transactions. There is still a potential for incentives here, because the beacon chain itself tracks ether, and transaction senders can offer a payment directly from their accounts as they stand before the block is created; we can assume by induction that those balances are validated. These blocks are subject to a round of consensus and formed into a chain.

- Stage 2:

Now that the transaction order is established, validators can go to work. Each one is responsible for following some contracts, and so is capable of validating transactions up to the point that one of them includes a call to a contract they don't follow and whose pre-state hasn't yet been validated. They release affirmations into the network for everything they can validate, and these get formed into a block, subjected to consensus, and committed to a second chain.

The second chain will necessarily lag behind the first one: there will be multiple Stage 2 blocks for each Stage 1 block until all validators can complete their work on the latter's transactions. But progress, however incremental, will always be made.

The best case is the existing transaction structure: noninteracting calls to separate contracts. Then this sharding will look just like it does now, all validators will be able to do their entire portion of the block at once, and the Stage 1 – Stage 2 correspondence is one-to-one.

The worst case is total interconnectivity, in which case the Stage 2 blocks have to be very short as validators coordinate their incremental efforts via the Stage 2 consensus process. This seems like an obvious and necessary limitation of scaling: if every transaction has global consequences, then either everyone executes everything, or everyone is waiting for everyone else to do their part. It's a basic concurrency tradeoff.

This execution structure would also create an opportunity to create smaller shards, simply by writing much more limited execution environments for special purposes, which may even end

and not require further following, or fragment into several successor environments (i.e. create new contracts and then self-destruct), each of which could then go to different validators. The validator pool could be much more flexible and, at least ideally, would not necessarily suffer from "horizontal" bloat of individual execution environments as they become too popular.

This post, once again, is very long and does not contain the kind of formal definitions that I see in Vitalik's work. I don't want to introduce ambiguity but I thought I could explain it better in a verbal narrative than through code. Hopefully anyone reading this will find that despite that, it's not entirely hand-waving.