At present, the bottleneck constraining throughput on the Ethereum 1.0 chain is state growth. So if we want to scale Ethereum, the logic goes, then 1000 shards where each has independent state would enable 1000x more throughput.

But consider the direction that Eth 1.x seems to be heading. The desire for Eth1.x is to make a large cost adjustment to two resource types: storage and tx data. Currently, storage is underpriced and tx data is overpriced. This incentivizes dapp developers to write contracts that utilize storage more than tx data, which results in storage becoming the throughput bottleneck. Proposals are to increase the price of storage, and decrease the cost of tx data. After these cost adjustments, developers will be incentivized to utilize tx data, and not storage (i.e. they will be incentivized to write stateless contracts rather than stateful). Thus in the near feature (if the Eth 1.x roadmap achieves adoption), we can expect that throughput on Ethereum 1.0 will be constrained by tx data, and not storage.

If we assume that throughput is constrained by tx data, then in order to scale Ethereum, shards on Serenity do not need to be stateful. If the bottleneck is tx data executed by stateless contracts, then 1000 stateless shards would enable 1000x more throughput.

Sounds great, but it requires shards that execute, which aren't planned until Phase 2. In the meantime, we can use Phase 1 as a data availability engine, a term that seems to be catching on. Let's think about how this will work.

Take the example of zk-rollup, which is constrained by data availability. Could a zk-rollup contract on eth1 make effective use of eth2 as a bridged availability guarantor? Well, if execution (i.e. verify the snark proof and update the state root) happens without a simultaneous data availability guarantee, then you have the plasma-ish zk-rollback, which gets you 17 zillion tps, but with a complexity tradeoff of needing plasma-style operator challenges and exit games. And in availability challenges, anybody can provide the data to prove availability, so its not really clear how putting the data in a bridged eth2 shard would simplify things.

Now with the other version of zk-rollup, i.e. the500 tps zk-rollup, everything is much simpler. Instead of needing a designated Operator, anyone can act as a Relayer at any time and generate snark proofs to update the state. The fact that a data availability guarantee always comes with every state update means that there are no plasma-style operator challenges and exit games to deal with. But it requires that execution happen in the same transaction as the data availability guarantee, and unfortunately we can't do that with a bridged availability engine. In other words, a bridge is sufficient for a fraud proof system like zk-rollback, but not a validity proof system like zk-rollup. So the important feature we need in an availability engine at Layer-1, in order to get the simplicity of validity proofs at Layer-2 is, apparently, the ability to guarantee data availability atomically with executing the state transition.

Maybe we should not be surprised at this realization. If data availability alone (with no execution) was truly useful, then there wouldn't have been talk about Phase 1 launching only to guarantee availability of a bunch of zero-filled data blobs, and there wouldn't have been dissatisfaction over having to wait yet another

launch phase before eth2 can actually do something useful (besides PoS). We're trying harder to use Phase 1 as a data availability engine, but it is still out of reach of any execution, so it feels underwhelming (Yay, we can do sovereign Mastercoin 2.0!).

So what are the reasons for resisting execution in Phase 1? Well, if we are assuming stateful execution, then everything revolves around each shard maintaining some local state. If validators are required to maintain lots of local state, then validator shuffling becomes a lot more complex. On the other hand, if we aren't doing execution then there's no local state to worry about. Validator shuffling becomes a lot simpler, and we can focus on constructing shard chains out of data blobs, and launch a lot sooner.

But let's not assume execution is stateful. What if we try to do execution with a stateless, dead simple VM?

Suppose there are three new validator fields in the BeaconState: code

, stateRoot

, and deployedShardId

. And there's a function, process_deploy

(right below process_transfer). When code is deployed, a validator must maintain the minimum balance (so at least 1 ETH is locked up. if there is no SELFDESTRUCT in the code, then 1 ETH is effectively burned and the code is permanently deployed).

Now there are accounts with code in the global state.

Next we try to get a particular data blob included in a shard, but how? As far as I know, it is an open question how shard validators in Phase 1 will decide what data blobs to include in shard blocks. Suppose that the Phase 1 spec leaves this unspecified. Then for a user to get their data blob included in a shard, they would either have to contact a validator and pay them out-of-band (e.g. in an Eth 1.0 payment channel), or they would have to become a validator and include it themselves (when they are randomly elected as the block proposer for a shard). Both of these are bad options.

A better way is to do the obvious and specify a transaction protocol enabling a validator to pay the current block proposer a fee in exchange for including their data blob in the shard chain. But if beacon block operations such as validator transfers have minimal capacity, then that won't work. Without a transaction protocol enabling validators to prioritize what data blobs they'll include, the "phase 1 as a data availability engine" use cases will be crippled (whether for contracts on eth1 using a bridge to the beacon chain, or Truebit, or Mastercoin 2.0, or any of the data availability use cases I've heard proposed). In any case, let's just assume that however shard proposers are deciding what blobs to include in the "data availability engine without execution" model, we are doing the same thing in a "data availability engine with dead simple stateless execution" model.

So a particular data blob is included in a block. Limit execution to one tx per block (e.g. the whole blob must be one tx). We're also not specifying whether the tx has to be signed by a key (if we have a transaction protocol), or if the tx is not signed (assuming no tx protocol). Let's assume the latter, with the code implementing its own signature checking (a la account abstraction; there is a block gas limit, but no fee transfer mechanism so no gas price and no GASPAY opcode). If the blob can be successfully decoded as a tx, then execute the destination account code with the data and current state root as input. If execution returns success, then the return data is the new state root.

How do we update the validator account stateRoot

? We can't update it in the BeaconState on every shard block (again, because of the strict limits on the number of beacon chain operations). But shard fields in the beacon state are

updated on crosslinks. Take the list of updated state roots for accounts on the same shard, and suppose they are hashed into a shard_state_root

. Seems not that different from the crosslink_data_root

(both are hashes dependent on the content in previous shard blocks) that is in Phase 1 already.

Admittedly, because all shard state roots are not updated every beacon block, there is some local state. But if accounts are global, then the state root data will be minimal. It seems not that different from the some number N of recent shard blocks that need to be transferred between validators during shuffling anyway.

Enough details have been glossed over. The point I'm trying to make is that the requirements for stateless execution seem to be mostly already satisfied in Phase 1. The biggest issue imo is the unspecified way that users will get their blobs included into the chain (which again, if not solved this issue may prevent Phase 1 from being usable even as a bridged availability engine). Or maybe its just the first issue, and I'm overlooking other big ones. What am I missing? What would be the most difficult part of bolting this onto Phase 1 (or Phase 1.1, if you prefer)?

The big reason for the simplicity of this execution model compared to Phase 2 proposals seems to be that contract accounts are global, like validator accounts. This means the number of contract accounts must be limited and so it will be expensive to deploy code in the same way it is expensive to become a validator (though hopefully not quite as expensive ;). But if we get to introduce execution into Eth2 much sooner, isn't this an acceptable tradeoff? Deployed code is equivalent to immutable contract storage, so another way to state what we're trying to do is to offer execution in Phase 1 without trying to scale contract storage. We still scale the important use case: massive throughput of data availability (1000x the transaction throughput).

Even with basic stateless execution, users can do cross-shard contract calls by passing proofs of state about one contract as the tx data to another contract. Contracts could also implement their own receipt-like functionality (a receipt in a contract's state root is just as verifiable as a receipt field in a block header). The developer experience is not great because there is no assistance from the protocol. But the Phase 2 proposals being circulated also seem to be lacking real features to facilitate cross-shard contract interaction (the messy stuff is left to the dapp developer, who must implement logic for getting receipts from different shards, making sure receipts are not double-spent, and so forth). So when it comes to developer experience, basic Phase 1 stateless execution does not sound much worse than the "simple" Phase 2 ideas. Basic stateless execution would also be sufficient to enable two-way pegs between BETH on the beacon chain and ETH on the main chain.

The main difference compared to Phase 2 proposals is that they aim to scale contract storage. But storage, and hence stateful execution, also seems to be the source of most complexity making it difficult to imagine including execution in Phase 1.