tl;dr: We mean a lot of things when we say “scalability”, and we can’t ignore the fact that by designing consensus to support a particular execution model scalably, we are missing design possibilities that could be realized by formulating a more ideal execution model. Ultimately, we may not even lose anything, and gain cool stuff like multi-chain communication and separate execution models.

Links that appear below:

- Phase One and Done
- Smart transactions

Now the long part.

I’ve been rereading @cdetrio’s Phase One and Done post and it has made me think about in what ways we are trying to improve scalability. At least to me (and since I am still figuring this out, probably only me), it seems like there are quite a few things that need to scale, but they are different and their needs are different.

1. Light versus full clients: a light client protocol is necessarily a kind of scaling solution, but there are different degrees of lightness and, if other aspects of scalability are achieved, there may not even be a difference between the two kinds of clients.
2. Protocol load: how much work is imposed on clients both light and full just to operate the blockchain (at the level of Phase 1: consensus, but no block content semantics).
3. Semantic initialization load: the work that a client needs to do after receiving the blocks, just to maintain some internal state that is aware of the nature of the block contents. This is pre-execution.
4. Transaction dependence load: the work of executing other transactions in order to prepare the historical state for individual transaction execution.
5. Individual execution load: the work of interpreting the semantics of the blocks’ data blobs, say in an EVM or (if this ever comes to pass) an alternate execution engine, assuming that the necessary historical state is present already.
6. Execution storage load: Keeping the historical state.
7. Protocol storage load: Keeping protocol elements, like blocks.

The light/full client dichotomy is particularly evident in #2

, initialization: a light client tries to avoid needing most blocks, which also reduces #3

(historical state is received as provable data blobs) and possibly #5

, though also possibly not. However, if any of these loads are reduced, or most likely all of them in any clearly superior scaling solution, every client is in part a light client. A way to reduce #3

without changing anything about the protocol is, of course, to introduce a separate zero-knowledge proof layer in order to distribute those blobs, and this does seem to me to be the only way to actually truncate

history traversals rather than merely separating

independent ones, so I’ll be talking about just stuff that a client can do completely on their own. Ultimately, someone does have to produce those proofs anyway.

Now, it seems to me that sharding is oriented towards reducing #1

and #5

, and addressing #2

. I say “addressing” because although shards do reduce the number of transactions any client needs to be aware of in the moment, the cross-shard dependency problem seems to be alive and well, which means that the work of #2

can’t finish and that of #3

can’t start for some transactions. It also creates a massive chasm between two kinds of clients (the validators and the rest). This might be affected by crosslinks in a way that I don’t understand yet.

I think there is a lot to say if we invert the priorities and try to reduce #3

and #4

, and see what demands that places on the protocol in order to achieve #1

. #2

will necessarily be informed by #3

, and new possibilities for #5

may open up if the relationship between execution and storage changes. Even better (to my mind), by putting the protocol last, implementation questions such as clever data structures (Merkle stuff) can be deferred, since clients, during execution, don't need to operate trustlessly with respect to themselves.

Coming back to @cdetrio's post (which I freely admit covers a lot of the same ground in different language and with more informed technical detail), I see a simple proposal: hijack the uninterpreted contents of Phase 1 blocks and, ideally without adding more state to those blocks, just use those blobs as stateless transactions in some execution model. This, of course, completely eliminates load #3

and nearly load #5

(unless I'm misunderstanding "stateless" here; I guess we are keeping accounts still, so there is Ether balance state). This clearly affects considerations for Phase 2, specifically by making it unnecessary to achieve execution. It also assists with reducing #4

, because stateless transactions are much easier to parallelize, i.e. separate into shards.

The proviso there is "unnecessary to achieve execution"; a Phase 2 would still be necessary for achieving a sharded EVM, with its global state. I personally don't like the way that execution model forces such a complex protocol solution in order to separate execution threads, but it is unarguable that it also permits a different

programming experience from the one that would result from following the suggestion I make in my previous post Smart transactions, and one that may make some kinds of contract logic more obvious. What I suggest there amounts to an expansion of stateless data-availability-based execution to execution that makes it much easier to organize state dependencies, so still reduces #3

, obviously adds back some of #5

, and still helps with #4

.

Given that some

execution model is possible right on top of Phase 1, it seems like Phase 2 can itself be built as a meta-protocol inside that one. This amounts to creating a new class of privileged transactions that don't need the same amount of history-tracking as a fully EVM-style transaction, and without actually sacrificing EVM transactions to do it. Validators can be concerned with checking the additional content of the blobs that isn't checked by the general consensus protocol, and in particular not

be concerned with checking the Phase 1 executions. There would be a straightforward way of allowing these two to interact (which would of course introduce some historical dependencies into the "stateless" portion of the chain), and by extension, allowing any two execution protocols built on top of the Phase 1 model to interact. This is a nice multi-chain solution; it is like writing the EVM inside another, simper to scale virtual machine that can also host other smart contract layers.

Finally, if this is done, it may allow us to improve sharding itself by reducing its support for the particular imagined execution model of the EVM. Sub-protocols built in the Phase 1 "VM" can then introduce further sharding protocols according to what those execution models permit. This gives us an avenue towards shards-with-shards.

I fear that my entire perspective on this may be hopelessly naïve in a way similar to the one that @cdetrio pointed out in his comment on his post (not in those words), simply by being unaware of the theoretical struggles of the past that led to rejecting certain superficially-appealing models. However, in the case of separate consensus and execution, both he and I seem to believe that some implementation may in fact be possible, so perhaps that ignorance isn't completely fatal.

I appreciate your dedication in reaching this point. Your comments are welcomed.