There exists a protocol transformation that theoretically can be made to many kinds of protocols that in mathematical terms looks as follows. Suppose that we use the state transition lingo, STF(S, B) -> S', where S and S' are states, B is a block (or it could be a transaction T), and STF is the state transition function. Then, we can transform:

S -> the state root of S (ie. the 32 byte root hash of the Merkle Patricia tree containing S)

B -> (B, W), where W is a "witness" - a set of Merkle branches proving the values of all data that the execution of B accesses

STF -> STF', which takes as input a state root and a block-plus-witness, uses the witness as a "database" any time the execution of the block needs to read any accounts, storage keys or other state data [exiting with an error if the witness does not contain some piece of data that is being asked for], and outputs the new state root

That is, full nodes would only store state roots, and it would be miners' responsibility to package Merkle branches ("witnesses") along with the blocks, and full nodes would download and verify these expanded blocks. It's entirely possible for stateless full nodes and regular full nodes to exist alongside each other in a network; you could have translator nodes that take a block B, attach the required witness, and broadcast (B, W) on a different network protocol that stateless nodes live on; if a miner mines a block on this stateless network, then the witness can simply be stripped off, and the block rebroadcasted on the regular network.

The simplest way to conceive the witness in a real protocol is to view it as an RLP-encoded list of objects, which could then be parsed by the client into a {sha3(x): x} key-value map; this map can then simply be plugged into an existing ethereum implementation as a "database".

One limitation of the above idea being applied to Ethereum as it exists today is that it would still require miners to be state-storing full nodes. One could imagine a system where transaction senders need to store the full state trie (and even then, only the portions relevant to them) and miners are also stateless, but the problem is that Ethereum state storage access is dynamic

. For example, you could imagine a contract of the form getcodesize(sha3(sha3(...sha3(x)...)) % 2**160)

, with many thousands of sha3's in the middle. This requires accessing the code of an account that cannot be known until millions of gas worth of computation have already been done. Hence, a transaction sender could create a transaction that contains a witness for a few accounts, performs a lot of computation, and then at the end attempts to access an account that it does not have a witness for. This is equivalent to the DAO soft fork vulnerability.

A solution is to require a transaction to include a static list of the set of accounts that it can access; like EIP 648 but much stricter in that it requires a precise enumeration rather than a range. But then there is also another problem: by the time a transaction propagates through the network, the state of the accounts it accesses, and thus the correct Merkle branches to provide as a witness, may well be different from the correct data when the transaction was created. To solve this, we put the witness outside

the signed data in the transaction, and allow the miner that includes the transaction to adjust the witness as needed before including the transaction. If miners maintain a policy of holding onto all new state tree nodes that were created in, say, the last 24 hours, then they will necessarily have all the needed info to update the Merkle branches for any transactions published in the last 24 hours.

This design has the following advantages:

1. Miners and full nodes in general no longer need to store any state. This makes "fast syncing" much much faster (potentially a few seconds).
2. All of the thorny questions about state storage economics that lead to the need for designs like rent (eg. https://github.com/ethereum/EIPs/issues/35 http://github.com/ethereum/EIPs/issues/87 http://github.com/ethereum/EIPs/issues/88) and even the current complex SSTORE cost/refund scheme disappear, and blockchain economics can focus purely on pricing bandwidth and computation, a much simpler problem)
3. Disk IO is no longer a problem for miners and full nodes. Disk IO has historically been the primary source of DoS vulnerabilities in Ethereum, and even today it's likely the easiest availability DoS vector.
4. The requirement for transactions to specify account lists incidentally adds a high degree of parallelizability; this is in many ways a supercharged version of EIP 648.
5. Even for state-storing clients, the account lists allow clients to pre-fetch storage data from disk, possibly in parallel, greatly reducing their vulnerability to DoS attacks.
6. In a sharded blockchain, security is increased by reshuffling clients between shards frequently; the more quickly clients are reshuffled, the more adaptive the adversaries that the scheme is secure against in a BFT model. However, in a state-storing client model, reshuffling involved clients download the full state of the new shard they are being reshuffled to. In a stateless client, this cost drops to zero, allowing clients to be reshuffled between every single block that they create.

One problem that this introduces is: who does

store state? One of Ethereum's key advantages is the platform's ease of use, and the fact that users do not have to care about details like storing private state. Hence, for this kind of scheme to work well, we have to replicate a similar user experience. Here is a hybrid proposal for how this can be done:

1. Any new state trie object that gets created or touched gets by default stored by all full nodes for 3 months. This will likely be around 2.5 GB, and this is like "welfare storage" that is provided by the network on a voluntary basis. We know that this level of service definitely can be provided on a volunteer basis, as the current light client infrastructure already depends on altruism. After 3 months, clients can forget randomly, so that for example a state trie object that was last touched 12 months ago would still be stored by 25% of nodes, and an object last touched 60 months ago would still be stored by 5% of nodes. Clients can try to ask for these objects using the regular light client protocol.

2. Clients that wish to ensure availability of specific pieces of data much longer can do so with payments in state channels. A client can set up channels with paid archival nodes, and make a conditional payment in the channel of the form "I give up $0.0001, and by default this payment is gone forever. However, if you later provide an object with hash H, and I sign off on it, then that $0.0001 instead goes to you". This would signal a credible commitment to being possibly willing to unlock those funds for that object in the future, and archival nodes could enter many millions of such arrangements and wait for data requests to appear and become an income stream.

3. We expect dapp developers to get their users to randomly store some portion of storage keys specifically related to their dapp in browser localstorage. This could even deliberately be made easy to do in the web3 API.

In practice, we expect the number of "archival nodes" that simply store everything forever to continue to be high enough to serve the network until the total state size exceeds ~1-10 terabytes after the introduction of sharding, so the above may not even be needed.

Links discussing related ideas:

- https://github.com/ethereum/sharding/blob/develop/docs/account_redesign_eip.md
- https://github.com/ethereum/EIPs/issues/726