TLDR:

Ethereum currently uses an accumulator (the Patricia-Merkle trie) which is designed for state. There's an alternative accumulator design which is a great match with the stateless client model, but it works for history only. By separating history and state, and encouraging use of history versus state, we can make the stateless client model more practical and scalable than initially thought.

History in Ethereum 1.0

In this post "history" refers to any append-only data structure relevant to Ethereum. We have transaction history, block history, receipt history. Contract code is also a form of history because contract code is immutable. Finally applications can exhibit history (e.g. think of a contract that maintains historical tick data in storage).

In Ethereum 1.0 history is a second-class citizen because transactions, blocks and receipts are not natively readable from transactions (with the exception of the block hash, but only for the 256 most recent blocks). Vitalik wrote 2.5 years ago that making history inaccessible "improves efficiency and code simplicity for many kinds of nodes" and that "state is all that matters". I imagine the rationale was that we cannot have nodes fetch potentially very old historical data to process transactions. Even today the folklore is that "ever growing blockchain history doesn't scale".

History in the stateless model

In the stateless model nodes never need to go digging for data (regardless of whether that data is history, state or something else). Indeed the responsibility of providing data is borne offchain (e.g. by the transaction sender). In that sense, history is no harder than state to process in the stateless model. Not only that, it turns out that handling history may be significantly easier

than handling state in the stateless model.

The key here is a recent innovation called asynchronous accumulators from Leonid Reyzin and Sophia Yakoubov. The specific construction in their paper is a clever and simple twist over the beloved Merkle tree; no magic. Their asynchronous accumulator has a property they call "low update frequency", which allows for history (without dynamism inherent to state) to be accumulated such that the witnesses for the individual events need only be updated logarithmically in the number of events (as opposed to linearly). And the cherry on top is that updates do not require knowledge of the accumulated set, perfectly matching the stateless model.

An asynchronous accumulator is useful for at least two reasons. First, it dramatically reduces the costs of witness maintenance for witness holders. Second and maybe more importantly, it greatly increases the probability that a transaction in the stateless model will be executable with the same witness data it was sent with (c.f. account lists). In other words, low update frequency might be the ingredient that makes the stateless client model practical.

Ethereum 2.0 idea: dual accumulators

So for Ethereum 2.0 we may want to try to flip around the philosophy of "state first, history second", instead aiming for "history first, state second" to leverage low frequency accumulator witness updates. Every contract would be endowed with a dedicated accumulator for its history, in addition to the existing read/write storage accumulator. Consider the following hybrid "dual accumulator" VM:

1. One accumulator for the history (keeping track of the "immutable past"). Append-only (non-dynamic); high bandwidth; low frequency witness updates; cheap; super large tracking set (think trillions of elements); implemented using an asynchronous accumulator

2. One accumulator for the state (keeping track of the "changing present"). Dynamic; medium bandwidth; high frequency witness updates; expensive; relatively small tracking set (think millions of elements); implemented similarly to the current Patricia-Merkle trie

My gut feel is that many (if not all) Ethereum applications can be tweaked to push almost all (80% to 99%+) of their storage load to the history (see below for a strategy to push all-but-256bits of the storage load to the history). Pushing data through the history accumulator would exhibit a greatly discounted gas price relative to pushing data through the storage accumulator, encouraging application developers to use storage only if absolutely necessary.

Writing "history-driven" applications

It turns out there are generic ways to write applications that maximise history and minimise storage. Below are two strategies:

1. Using SNARKs (or STARKs), it's possible for any application to use just 256 bits of storage, pushing everything else to history. The 256 bits of storage would hold the hash of the (implicit) state derived from the "transactions" pushed to the history, themselves proved valid with a SNARK. Each transaction would update the 256 bits of storage. The state is implicit because it is derived offchain from the "transactions" (a bit like Bitcoin balances are implicitly derived offchain from UTXOs). Notice that data availability for the implicit state is handled by the gossiping of historical transactions at the time of execution.

2. As an alternative to SNARKs/STARKs above, we can use a TrueBit/fraud proof model with potentially-invalid-but-collaterised transactions (pushed to the history) that “confirm” after a given period of time during which no successful challenge was made. The application would use storage as a buffer for temporarily unconfirmed transitions state of the implicit state.