State size management in Ethereum

Framework for change plans

Outline - questions in sequence

- Why is state a valuable resource and to whom? (can this value be replicated)
- Why state size needs to be managed? (effect on system performance + possible partial mitigations)
- How can state size be managed? (feedforward vs feedback control)
- What metrics need to be regulated? (state size, state growth, or both)
- Do metrics need to be in-state or off-state?
- What parameters (control inputs) do we use? (cost of state expansion, charge per size unit per block)
- Can controlled system weaken or evade control? (dark rent, miner rebates)

Why is state a valuable resource and to whom?

Ethereum state provides these resources:

1. It keeps pieces of data (e.g. established claims, like a token balance) automatically available and their authenticity provable at any time. In this case, **value is provided to the beneficiaries of the claims**.
2. It enables ad-hoc permissionless stateful communication (i.e. beyond the boundaries of 1 transaction) across contracts (synergy), a higher order resource. Example is the synergy between DAI token system and decentralised exchanges. In this case, **value is provided to the users transacting via such synergies**.

Why state size needs to be managed?

In this framework, we choose to ignore the cost of running a full node (cost of storage etc.).

We only looking at the state size from the perspective of Ethereum blockchain along different dimensions (reading data from the state, sealing the blocks - computing Merkle state root, snapshot sync).

Why state size needs to be managed?

Placing information into the Ethereum state, while providing value to users, leads to degradation of performance of the the Ethereum nodes:

1. Slower transaction processing (verification) due to reading from the state

2. Slower block sealing (verification) due to construction of the new state root after modifications

3. Longer snapshot synchronization due to bandwidth consumption to transmit the entire state

4. Lower success rate of snapshot sync due to state pruning by the peers

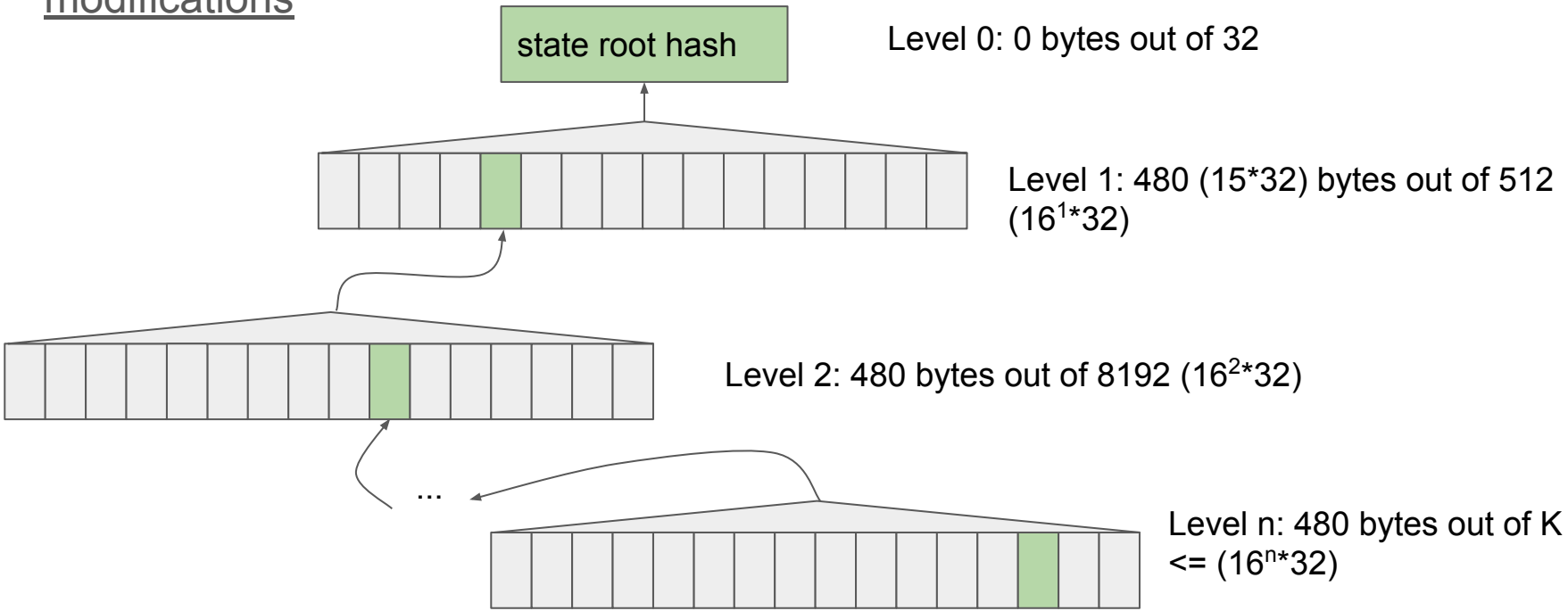
Why state size needs to be managed? (1)

Slower transaction processing (verification) due to reading from the state

Theoretically, if state is stored as key-value pairs (like in Turbo-Geth), transaction processing is slowed down as log (due to the search in the database index) of the state size. If, instead, the state is stored as a trie (like in geth, Parity, etc), the **transaction processing is slowed down as $\log^2$ of the state size** (one log due to number of levels in the trie, another log is due to the search in the database index), and **linearly with the number of unique items accessed in a block of transactions**.

Why state size needs to be managed (2)

Slower block sealing (verification) due to construction of the new state root after modifications



Why state size needs to be managed (2)

Comments to the previous slide:

To reconstruct merkle root of the state, it is required to know all the sibling hashes in the path from the state root to the modified item. This means fetching 480 bytes for each tree level (there are 15 sibling hashes on each level, 32 bytes each). On average, it could be 7-8 levels - $\log_{16}(\text{state\_size})$, meaning 3840 bytes per item. Some of these 3840 bytes will be the same for some modified items. More reasoning and simulation is needed, but it can be expected that modifying 1 items in the state requires fetching about 2k of data out of the state trie. Therefore, **it slows down block processing as $\log^2$ with the size of the state and linearly with the number of items modified in a block**.

Why state size needs to be managed (2)

Comments to the previous slide:

Caching can be employed to reduce the performance impact of state size on state access. However, any non-randomised caching policies (for example, LRU - Least Recently Used) can become DoS vectors, because the adversary can construct transactions to specifically generate cache hits. Only randomised cache (where cache keeps a random sample of the state) would make the network as a whole free from such vectors.

Therefore, caching cannot reduce the $\log^2$ function, but only slightly reduce the coefficient.

Why state size needs to be managed (3)

Longer snapshot synchronization due to bandwidth consumption to transmit the entire state

General idea of snapshot sync is to transmit the entire state from a group of peers to a new node. Depending on the scheme, state is authenticated either once at the end of the sync (warp sync), or piece by piece (fast sync).

Intuitively, **sync duration grows linearly with the size of the state**.

Why state size needs to be managed (4)

Lower success rate of snapshot sync due to state pruning by the peers

With the growing state size, most nodes start pruning aggressively.

If a new peer initiates a fast sync (downloading the state), and does not complete within 1 hour, all peers that only keep last 256 state snapshots (256*15s ~ 1h), would have pruned the data necessary for the new peer to complete the sync. So it has to restart the sync with the risk of never completing.

Intuitively, **rate of snapshot sync success would be highly non-linear**, and dependent on the ratio between prevailing network bandwidth and prevailing pruning threshold. The larger is the state, the lower is the threshold.

Possible partial mitigations

It might be possible to mitigate some of the performance effects of the state size with short terms measures.

- Latency (but not throughput) of block processing can be improved by pre-announcing “half-mined” (half of PoW difficulty) blocks, letting nodes to warm up the state caches for when the fully mined block arrives.
- All state items (accounts and storage items) can include “last modified block” field, allowing more elaborate syncing protocols that do not need to start from scratch when the sync fails.

Why state size needs to be managed?

How do we measure this:

4. **Lower success rate of snapshot sync** due to state pruning by the peers

Emulation with the goal of learning the the function for various sync mechanisms (fast sync, warp sync, fast warp sync, others):

```
sync_success_rate = func(state_size, bandwidth, pruning_threshold)
```

Why state size needs to be managed?

How do we measure this:

1. **Slower transaction processing** (verification) due to reading from the state

2. **Slower block sealing (verification)** due to construction of the new state root after modifications

3. **Longer snapshot synchronization** due to bandwidth consumption to transmit the entire state

Emulation with the goal of learning the coefficients for these performance degradation functions

Why state size needs to be managed?

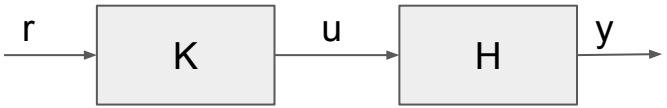
How do we measure this:

Once we know the functions (and roughly coefficients), simulation can be used to estimate at which state size we are likely to hit events like this:

- New geth nodes mostly unable to snapshot sync
- New parity nodes mostly unable to snapshot sync
- Block processing takes 10% of interblock time (1.5 seconds) on average, even with the current block gas limit and pattern of usage (we learn pattern of usage from mainnet data)
- Block processing takes 50% of interblock time (7.5 seconds) on average.

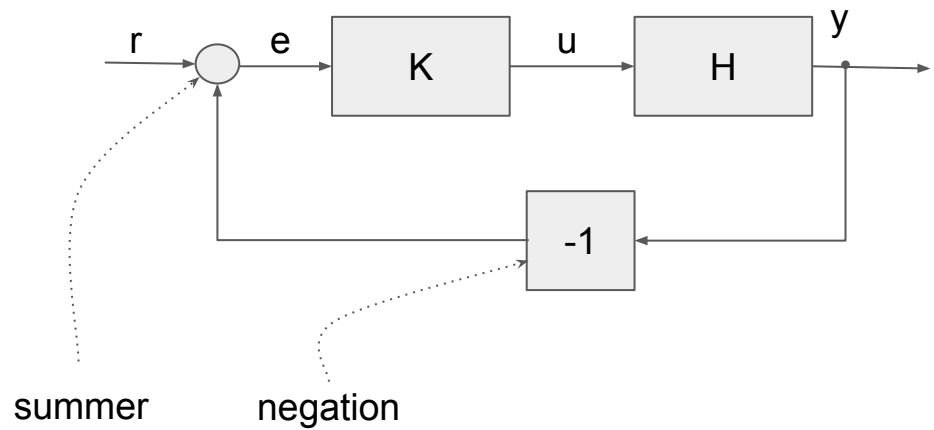
How can state size be managed?

Feedforward control (open loop)

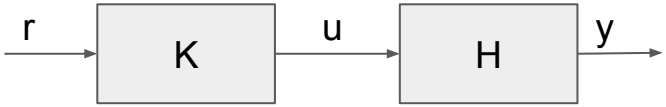


K - controller
H - controlled system
y - regulated output (metric)
r - reference value (parameter)
u - controller output
e - tracking error

Feedback control (closed loop)



Feed-forward control

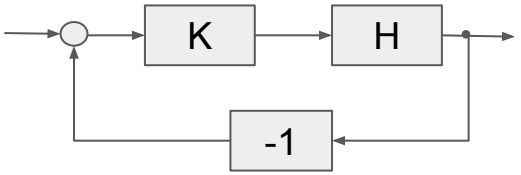


Would be useful if we wanted to regulate something that the protocol cannot observe within itself, for example, latency/throughput of block processing, snapshot sync duration, or snapshot sync success.

Gas schedule of EVM instructions is an example of feed-forward control - the gas costs get adjusted in the hope to eliminate mispricings, which are not observable from the protocol currently.

Obvious drawback of feed forward control is that is can be unreliable, especially if the controlled system (i.e. Ethereum network) keeps evolving, and requires regular changes in the controller (via hard forks).

Feedback control



Obvious example of feedback control in Ethereum is mining difficulty, and it is possible because protocol can perform time measurements using block timestamp.

If we wanted to use feedback control, we would mostly probably target not the performance degradation effects directly, but the state size (or state size growth rate) as a proxy.

What metrics needs to be regulated?

If we choose to use Feed Forward control, we should be targeting the performance effects directly (measured under some standardised benchmarks):

- Block processing latency
- Block processing throughput
- Duration of snapshot sync (via network simulation/emulation)
- Success of snapshot sync (via network simulation/emulation)

If we chose to use Feedback control, we should be targeting size of the state, or a form of moving average of state growth, or both.

Do metrics need to be in-state or off-state?

For feed-forward control, it is obvious that the metrics do not need to be in-state.

For feedback control, the state size (or state growth) needs to be embedded into the state. This is to ensure that the correctness of controller can be verified for any block in isolation (plus some number of previous blocks for moving average).

State growth metric is easier to introduce into the state - it requires an additional field in a block (or in the state under the state root hash). The value of this field can be computed efficiently when processing a block.

State size can be introduced in 2 stages (hard forks): 1) cumulative state growth starts to be recorded. 2) state size at hard-fork 1 is added.

What parameters (control inputs) do we use?

We know what type of actions change the state size (and state growth rate):

- Creation of an account by sending non-zero ETH to an empty
- Creation of a contract by transaction without destination but with data
- Creation of a contract from another contract using CREATE or CREATE2
- Allocation of a new storage item using SSTORE
- Removing a storage item by setting it to zero using SSTORE
- Removing contract by using SELFDESTRUCT

First 4 increase state size, last 2 decrease state size

What parameters (control inputs) do we use?

Since we must not be able to force users to stop making transactions at the times of our choosing, there are 3 possible ways to control state size and state growth

1. Make state increasing actions more costly
2. Make state decreasing actions more rewarding
3. Reduce the state directly (eviction)

What parameters (control inputs) do we use? (1)

Make state increasing actions more costly

An obvious way of making actions more costly is to increase their gas cost via hard fork. Such an increase is supposed to have 2 effects - scarcity and cost:

- If the block gas limit does not increase, there can potentially be fewer state expansions per block. For example, if block gas limit = 8M, and cost of new storage item is 20k, there can only be 400 new storage items per block. With new item costing 80k, there can only be 100 new storage items per block.
- Senders of transactions will expand state less if it cost more gas (?)

What parameters (control inputs) do we use? (1)

Make state increasing actions more costly

The price paid for gas is determined by only two parties: transaction sender and the miner. The gas price recorded in a transaction might not be the actual gas price paid from the sender to the miner, due to possible private agreements. Therefore, with some work and organisation, senders can evade the second effect of the gas cost increase.

Open question: can we ignore the evasion of the 2nd effect and only rely on the 1st?

What parameters (control inputs) do we use? (2)

Make state decreasing actions more rewarding

There exists a system for rewarding state decreasing actions - gas refunds. However, these refunds needed to be (somewhat arbitrarily) capped by a half of gas consumed, to make sure miners still mine transactions with refunds. This capping greatly reduces the incentive for clearing the state.

What parameters (control inputs) do we use? (2)

Make state decreasing actions more rewarding

To make refunds simple and more effective, they need to be given by the protocol and not by the miners. This logically means that:

a) state expansion needs to burn (lock) ETH so that state expansion-clearing cycles cannot be used to issue new ETH
b) amount of ETH burned (locked) at state expansion needs to be more or equal to the amount of ETH released as a refund - this precludes denominating burning and refunding in gas

What parameters (control inputs) do we use? (2)

Make state increasing actions more costly & make state decreasing actions more rewarding

Incidentally, requiring to burn (lock) ETH at state expansions, strengthens the control, because now the state expansion transaction has 3 parties - sender, miner, and all ETH holders.

Since we cannot use gas price for the cost of state expansion, it need to be expressed in ETH. Clearing pre-existing state items should not produce new ETH.

What parameters (control inputs) do we use? (2)

Make state increasing actions more costly & make state decreasing actions more rewarding

Possible designs:

- Constant amount of ETH is burnt (locked) at expansion and refunded (released) at clearing. All new items get a flag indicating there is a locked ETH
- Amount of ETH burnt and refunded depends on the state growth rate or state size (via feedback control loop), and all state items “remember” the amount locked in them.

What parameters (control inputs) do we use? (3)

Reduce the state directly (eviction)

Obvious questions to ask here are:

- What (which accounts and storage items) to evict?
- When to evict?

What parameters (control inputs) do we use? (3)

Reduce the state directly (eviction): What (accounts, storage items) to evict?

Intuitively, we should evict something that “no one cares about”, or, “no one cares about enough”. There are two ways the protocol can attempt to figure this out:

1. When was the item last queried or modified
2. How much ETH someone is prepared to pay to keep item in the state (because it provides value as mentioned early)

There are blind spots, of course. Some important things are rarely accessed, and no one would initially be prepared to pay upkeep. Therefore, evictions need to be reversible

What parameters (control inputs) do we use? (3)

Reduce the state directly (eviction): What (accounts, storage items) to evict?

Important invariant that must be preserved upon evictions is that contract storage cannot be partially evicted. If a contract has a large storage, this means that it either has to stay in the state entirely, or be evicted entirely. Accessing such large contracts in order to refresh their timestamps and avoid eviction, becomes a very cheap way to neutralise the control. Therefore, evicting only on the basis of the last access time would be a very weak control.

What parameters (control inputs) do we use? (3)

Reduce the state directly (eviction): What (accounts, storage items) to evict?

Eviction policy based on “How much ETH someone is prepared to pay to keep item in the state” leads to the concept of State rent. Protocol charges all items in the state some amount of ETH every block. Eviction happens when the rent charge is not paid.

Can controlled system weaken or evade control?

What happens to the state items that existed before the control was introduced? Any control in systems like Ethereum has to be introduced with a lead time of at least few months. If such control exempts anything that existed before the introduction, there could be an incentive to exploit this window between the announcement of control, and its implementation.

In the case of state size control, if we make state expansion more costly after date Y, and announce it on the date X, then during the time period (X;Y), it could be potentially profitable to “hoard” state items, to be able to sell them after the date Y. This needs to be taken into account in any proposal.