I assume just a single shard with one global account state represented by one Merkle-Patricia Trie (MPT), as originally proposed in the yellow paper. I am aware of Verkle tries, and this idea seems orthogonal to it. Also, I am pretty familiar with Aleph C++ implementation (currently deprecated), while I checked geth only very briefly.

Idea

The global state as an instance of MPT is a data structure with exclusive access - meaning that only a single tx can be processed at one time. Processing of a single tx is expensive regarding storage/memory accesses - descending to a leaf node with an account state to update it might require several storage accesses (even though the caching might help to some extent) + the same number of write modifications. From some of my experiments made within a small research project Aquareum, this looked as the biggest bottleneck of EVM execution. Anyway, I was thinking why not to replace a single exclusive MPT by a number of independent MPTs that would enable parallel processing of txs, while all these small MPTs could be aggregated by a standard binary Merkle tree after all txs of a block have been processed in EVM already - root hash would be stored in a header instead of MPT root, so light clients would not lose any integrity information.

Details

The number of such MPT should respect requirements for parallelization (no. of cores/threats, distribution of no. of txs modifying more account states, etc). For example, it could consume 3 nibbles of the original key to a single global MPT, representing 2^12 = 4096 small MPTs, while the path in small MPTs would start addressing from the 4th nibble of the key. In this way, txs modifying just a single account state could be heavily parallelized but txt modifying more than 1 account state would need a lock on all account states being modified. They could be known beforehand (w/o executing tx), in which case the planning should be trivial but in some cases, they might be known only dynamically (while executing tx's code). In the latter, dynamic synchronization primitives of process scheduling could be used and probably it would involve some small overhead which, however, should be compensated for still interesting parallelization.