This is a continuation on booster rollups.

The previous post was about how we can leverage booster rollups to scale storage and transaction execution in a general and sane way. In this post, we describe another possible use case where the rollup is only used to help scale transactions, while keeping all state on L1. This effectively makes it a coprocessor with very few hard scalability limitations.

But first, a recap, because some definitions have changed:

Definition

Booster rollups are rollups that execute transactions as if they are executed on L1, having access to all the L1 state, but they also have their own storage. This way, both execution and storage are scaled on L2, with the L1 environment as a shared base. Put another way, each L2 is a reflection of the L1, where the L2 directly extends the blockspace of the L1 for all applications deployed on L1 by sharding the execution of transactions and the storage.

Booster rollups allow scaling a chain in many ways. A single booster rollup instance can be used as a fully independent EVM environment, a ZK coprocessor, or anything in between, simultaneously.

New precompiles

- L1CALL: Allows reading and writing L1 state.
- L1SANDBOXCALL: Allows reading and writing L1 state, but at the end of the call the L1 state changes are reverted.
- L1DELEGATECALL: Execute a smart contract stored on L1, but all storage reads and writes use the L2 state.

These definitions use L1, but in practice it just refers to the state of the parent chain.

Booster rollups as ZK coprocessors

Booster rollups allow all L1 smart contract work to be offloaded to L2 using a ZK-EVM, while keeping all state on L1. The only work required on L1 is verifying the ZK proof and applying the final state updates back to the L1 smart contracts. This allows using a booster rollup as a ZK coprocessor for all smart contracts of the parent layer, and of course also for one or more specific smart contracts. For example, it's possible to have a booster rollup for the whole L1, while also having additional booster rollups as ZK coprocessors on a dapp-by-dapp basis on that L2. Of course, in general, the more things can be batched together, the more efficient.

This is of course quite similar to other ZK coprocessors, likezkUniswap and Axiom, except in those cases some specific functionality is handled offchain. In booster rollups, the same L1 environment is maintained no matter where a transaction is executed. This means there is additional overhead on the proving side because the logic is not written in the most efficient way, however having a single general solution that is usable by all smart contracts with minimal work seems like an interesting tradeoff, depending on the usecase. There is certainly still a reason to optimize certain tasks as much as possible, so they are complementary.

One other use case that is also interesting is to use this method not for L1, but on a new layer that would be shared between multiple L2s just like we used the L1 state for shared data. This is a bit more flexible because we can have more control over this shared layer. So we can do things like automatically having all smart contracts have the applyStateUpdates

functionality, have a way to expose the latest shared layer state root to the EVM and allow the shared layer to be reverted when necessary (allowing state updates to be applied more optimistically with the zk proof coming later).

Implementation

We achieve this by (re)introducing the L1CALL precompile on L2. L1CALL executes transactions against the L1 state and storage writes are applied like they would on L1, just like L1SANDBOXCALL. Unlike L1SANDBOXCALL where all storage writes are thrown away when the call ends, we keep track of the resulting L1 state across all L1CALLs while also recording all L1 storage updates that happened during those calls in a list. If it's the first time a storage slot for a smart contract is updated, (contract, storage_slot) = value

is added to this list. If (contract, storage_slot)

is already in the list, the value is simply updated with the new value. The booster rollup smart contract then uses this list containing all L1 state changes to apply these changes back to the L1 smart contracts which were modified by the L2 transactions:

function applyStateUpdates(StateChange[] calldata stateChanges) external onlyFromBooster { // Run over all state changes for (uint256 i = 0; i < stateChanges.length; i++) { // Apply the updated state to the storage bytes32 slot = stateChanges[i].slot; bytes32 value = stateChanges[i].value; // Possible to check the slot against any variable.slot // to e.g. throw a custom event assembly { sstore(key, value) } } }

Smart contracts that want to support this coprocessor mode need to implement this function in their L1 smart contract.

The efficiency of this is great for smart contracts where only a limited number of storage slots get updated by a lot of L2 transactions (think for example a voting smart contract), or where certain operations just require a lot of logic that is expensive to do directly on L1. The amount of data that needs to be made available onchain is also limited to just the state changes list in most cases.

Limitations

There are some limitations we cannot work around (at least on L1), because some state changes cannot easily be emulated.

- Nonces can only change by doing a transaction from an account, and so we cannot set those to a specific value on L1.
- ETH, which is directly tied to the account on L1, cannot be changed using an SSTORE. This means that msg.value

needs to be 0 for all transactions passing through L1CALL.

- Contract deployments using CREATE/CREATE2 are also not possible to do using just SSTORE. It is technically possible to support them when we handle them as a special case however.

Replay protection

Replay protection of the L2 transactions is an interesting one. The L2 transactions can execute transactions directly against the L1 state, but it is not possible to also update the nonce of an EOA account on L1 without doing an L1 transaction (replay protection works fine with smart wallets that implement the applyStateUpdates

function). This means that the L2 transactions do need to use the nonce of the account using L2 storage, which is the only thing that prevents the ZK-EVM coprocessor to work completely stateless (excluding the L1 state of course). It is possible to work around this by requiring users to execute an L1 transaction to create an L2 transaction, but that would make transactions much more expensive and greatly limits the possible scalability improvements.

Fee payments

Fee payments for transactions can be done on L2, which would be the most general and efficient way. It could also be done by taking a fee from the user in the smart contract when possible (e.g. a swap fee for an AMM). Of course, that may decrease the efficiensy if it requries an additional L1 state change.

DA requirements

This L1 state delta is the only data that is required to be pushed onchain. However, to be able to create L2 blocks/transactions, it is required to know the nonces of the accounts on L2, but the security of the system does not depend on it (because all state is still on L1).

If fee payments are handled on L2, this data would also need to be made available onchain, though if the L2 balances are only used for paying fees, the balances will be low so the risk is also low.

Synchronization

The input for the booster rollup is the L1 blockhash of the previous L1 block. This blockhash contains the L1 state root after the previous L1 block. This is currently the most recent state root available in the EVM. This means that the ZK-EVM coprocessor needs to run as the first transaction touching the relevant L1 state in an L1 block, otherwise the state we execute the transactions against is outdated. This can easily be prevented, but it does limit the flexibility if both L1 and L2 transactions need to be combined for some reason. This inflexibility would be solved if there would be a way to expose the current L1 state root to the EVM.

Mixing and matching L1 and coprocessor blocks also requires the immediate application of the state changes, otherwise the L1 transactions would execute against outdated state. This prevents us from optimistically applying the state changes from L2, and so we need the ZKP immediately. This can be solved by using an intermediate shared layer instead of working directly on L1, where we would be able to revert when the block data is invalid. If no L1 transactions modify the same state as the L2 transactions, the state delta’s can be applied to L1 with a delay.

Chaining rollups

The updated L1 state root after each L2 block is exposed as a public input. This allows multiple L2s to work together on the latest L1 state, as updated by earlier L2 blocks. This is helpful to scale work that is not easily parallelizable over multiple L2s by splitting up the work over multiple rollups (though the execution is still sequential of course). For example, it is possible to do AMM transactions on rollup A, which all update the state of the pool, and then have rollup B continue against the state after rollup A, with finally the latest AMM state being applied back to the L1 smart contract. This makes it a convenient way to share shared sequential data across L2s.