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.

What

Rollups are commonly seen as their own separate chain, being almost completely independent from their host chain and their peer rollups. Designs have been [proposed](#) to bring L1 data to the rollup in an easy way, but on its own that still fails to create a uniform environment that can scale Ethereum in a convenient way. If the future demands hundreds or even thousands of rollups to adequately scale Ethereum, having each rollup function as an isolated unit, with its own set of smart contracts and rules, is not ideal. Developers would have to copy-paste their code onto each rollup. Instead, a rollup could directly add extra blockspace to all dapps deployed on L1. Deploying a rollup could be something like adding extra CPU cores and an extra SSD to a computer, with applications capable of taking advantage of these automatically doing so. Similar to multi-threaded applications, the dapp does need to be aware of this to be able to take full advantage of this multi-rollup world.

Advantages

- increases scalability in a transparent way

: Similar to adding extra servers to a server farm, the only thing required is adding an extra rollup and all applications can take advantage of the increased scalability, no extra steps required.

- uniform:

L1 and all L2s look and feel exactly the same for users. All smart contracts automatically have the same address across L1 and all booster rollups.

- easy for developers:

No need to deploy to all L2s that need support for the dapp. Just deploy your smart contracts to L1 and you're done. Each dapp is multi-rollup out of the box. Rolling out updates can now also be done in a single place, with for example all L2s automatically following the the latest version on L1.

- easy for users:

A single address everywhere, automatically, no matter if it's an EOA or smart wallet. Each smart wallet is automatically an L1 and multi-rollup smart wallet that the user can use to transact on L1 and on all L2s.

- easy for rollups:

No need to try and get app developers to deploy their dapp on your rollup. All dapps are automatically there (though it still requires to get developers to do some minimal offchain work to make it easily accessible to users, like UI things and changing RPC URLs). The goal now shifts to getting developers on board to write their dapps in the best way to take advantage of multi-rollups.

- stackable

: Combine a booster rollup with a based rollup and a way to do atomic cross-rollup transactions between all L2s in this booster network, and we're doing some serious Ethereum native scaling which I like to unironically call The Singularity and you can't stop me. This combination should get very close to the feeling of a single scalable chain for users. Not all L2s in this shared network need to be booster rollups, they can be combined with non-booster rollups as well.

- sovereignty

: No need for rollup specific wrapper contracts for things like tokens, each smart contract runs on L2 exactly the same way on L1 in all cases, the original developer remains fully in control.

- security

: No more rollup specific implementations to bridge functionality over from L1 also means no single point of failure anymore (like bridges with a shared codebase where a single hack can be catastrophic). The security is now per dapp.

- simple:

For rollups that are Ethereum equivalent, the only additional functionality to be a booster rollup is to support what the L1CALL/L1DELEGATECALL precompiles do in some way.

Disadvantages

- contract deployments need to be disabled on L2:

It needs to be ensured that the L2 keeps mirroring the L1 in all cases (so SELFDESTRUCT is also a no-go, but that is already going away). For that to be true, contracts can only be deployed on L1, which also makes sure all L2s have access to it. Note that this is not really a big limitation because this doesn't mean that each L2 needs to be behave exactly the same everywhere. It's perfectly possible for smart contracts to behave differently depending on which L1/L2 the user is interacting; it just needs to be done in a data driven way. For example, the address of the smart contract being called by another smart contract could be stored in storage. Because storage can be different between L1/L2s, the behavior of this smart contract can vary depending on which chain it is being executed on.

- contract code and shared state is still on L1

: The L1 is still used for the shared data, and so there is no direct increase in scalability for this. But that seems like an inherit limitation of any scalable system, it's up to app developers to minimize this as much as possible.

- not all dapps are parallelizable

: Similar to normal applications, not all of them are easily parallelizable in which case they cannot take full advantage of the shared/seperate storage model. But that's okay, smart contracts like this still scale on all the different L2s separately which is the status quo. And there's still the big advantage of the smart contract being available on all L2s automatically. This is also why it's still very important that users can seamlessly do transactions with smart contracts on any of the L2s in the network, no matter where the transaction originates, because e.g. some dapps may run their main instance on a specific L2, or have the most liquidity available there (like the uniswap pool for a specific trading pair).

- L1 and L2 nodes have to run in sync, with low latency communication:

Booster rollups basically are the L1 chain, they just execute different transactions and have some additional storage of their own. A possible implementation could be to actually run both L1 and the L2 in the exact same node with just a switch deciding to use either the shared L1 storage or the L2 specific storage while executing the transactions.

How

All accounts on a booster rollup would have a fixed smart contract predeployed to them:

contract L2Account { fallback() external { // Check here if the smart contracts implements the expected interface. // If not, default to parallelize everything. // Check if the function being called supports parallelization if (address(this).l1call(isParallel(msg.sig)) { // Execute the call with the L2 state address(this).l1delegatecall(msg.data); } else { // Execute the call with the L1 state address(this).l1call(msg.data); } } }

(Note that EOAs are not handled in this code)

Each smart contract decides for itself (using the code deployed on L1, which optionally implements a very easy interface, otherwise it falls back to using the L2 state exclusively so the dapp runs completely on L2) which parts of its code need to be run with the state stored on L1 (L1CALL) which need to run with the state stored on L2 (L1DELEGATECALL). The state stored on L1 is the shared state, the state stored on L2 is the state that can be parallelized (e.g. token balances or specific uniswap pools).

Handling EOAs correctly here is challenging without additional precompile magic. Using this implementation using standard smart contracts also changes the gas cost of the execution compared to L1, so it is not ideal. In practice this logic would probably not be done in a smart contract, but would be built into the logic of the rollup instead so that it can be made fully transparent.

A simple example to make sense of this: for a token contract, the total balance would be stored on L1, but all user balances and transfers would be done in parallel on L1 and all L2s.

Booster enabled rollups

Any non-booster rollup that supports the L1CALL/L1DELEGATECALL precompiles also supports boosting, but that now requires deploying this smart contract to each L2 manually per dapp.