Hi Community,

We would like to share our ideas on extending solidity and EVM toward execution sharding.

Abstract

We extend the Solidity with a distributed programming model that enables general smart contract programming on multi-chain systems with execution sharding. The extended syntax allows developer to define:

- How contract states are distributed over multi-chains,
- How transaction workflows are separated and executed by relaying across multi-chains.

Motivation

A smart contract is defined equivalently to a sequential state machine. It implies a simple but constrained programming model.

- Sequential Execution

. Each contract function invocation or every transaction must be executed sequentially to avoid potentially unsafe concurrent access of ledger states.

- Single-Box States

. Since any function has direct and immediate access to any part of the states, the entire ledger states must be available in all nodes and kept synchronized as chain grows. This requires, at least, that all transactions making changes to ledger states must be transferred and executed in every node to keep the ledger states correctly updated.

Such a constrained programming model makes development relatively simple, equivalent to programming a single-thread CPU and to fit everything in a single-box computer. However, it is capped by the computing resource a single computer may have.

Dividing and distributing the workload to multiple computers is a time-tested design philosophy to achieve the scalability of a computing system. As for blockchain systems, leveraging multiple blockchains and distributing workload of the entire network across different instances of blockchains is the fundamental solution in the long run, so that the infrastructure is able to scale continuously as the crypto ecosystem grows.

In this proposal, we propose to extend Solidity and EVM to support multi-chain systems with execution sharding. The host model we are using is a system whose transaction executions and ledger states are divided and distributed across chains. We separate the strategies for dividing and distributing the workload from the underlying blockchain system’s architecture and employed consensus protocols.

Assumption of the Multi-Chain System

We aim to maximize generality in supporting different design of execution sharding blockchain system. While, at minimum, we assume that each chain of a sharding system has the following essential components:

- A Chain of Blocks

generates new block periodically with fixed, or fixed expectational, interval. Each block carries an ordered listed of unique transactions with limited total data sizes, or total computation cost for execution (e.g. Gas).

- A Storage

for Ledger States provides efficient immediate read and write of state data.

- An Execution Engine

runs deployed smart contracts as invocations made by transactions and updates ledger states according to execution outputs.

- A Mempool

stores, in memory, unconfirmed transactions to be picked up by future new blocks.

- A Broadcast Network

is a peer-to-peer network that replicates legitimate transactions and blocks across nodes.

Each chain has its own uniquely dedicate instances of the above five components. In addition, we have the following assumptions:

- Non-overlapping Workload

: Any transaction will be, and only be, confirmed and executed by a single chain. Each address or piece of ledger state will be hosted and updated by only one chain.

- Deterministic Distribution

: A transaction, an address or any piece of ledger state will always be associated with a specific chain, given a particular configuration of the multi-chain system.

- Proven Relay

: A relay proof can be generated from a block on one chain and verified by another chain when the corresponding relay transaction is received.

- Voluntary Relay Broadcast

: A relay transaction with correct proof will be voluntarily replicated across nodes and stored in the mempool like normal transactions.

Specification

To support the execution sharding on a multi-chain system, we introduce a new scope class specifier

that allows the developer to programmatically define how the states of a contract are partitioned into so-called scopes

and distributed across the system in multiple shards. A new relay statement

is added as a generalized way of expressing cross-chain workflow in smart contracts. This new programmable and flexible way enables any function invocation executed on one chain to trigger one or more subsequent asynchronous invocations of functions in other chains. We also add a couple of new opcodes to EVM to support these new features at the execution engine level.

Partitioning State Variables Using Scopes

A Scope

can be seen as a subset of a smart contract's state that is stored on the same chain in a multi-chain system. The states of a smart contract are decomposed into scopes with different synchronization requirements within- and across-chains/shards, allowing state sharding for general smart contracts. Scopes correspond to the data partitioning structures in sharded blockchains.

When declaring a state-variable, the developer can now specify an optional scope class

, as shown in the following examples:

uint @address balance;

In the above example, the state variable balance

is declared on scope class address

. It now has an individual instance for each possible value of address

type.

A scope class always starts with the '@' symbol, followed by any elementary type

or a reserved special scope name (discussed later)

Special scopes

Two new keywords global

and shard

are added to specify special scopes:

- @global

is the single scope where state variables are not partitioned and effectively singletons, this is also the default if no scope class is specified when defining a state variable. These state variables are shared and synchronized across all shards.

- @shard

is used for common contract states within a shard / chain in multi-chain systems. A shard scope state is instantiated and updated independently in each shard. For single-chain systems, it is equivalent to @global

.

From the perspective of object allocation, the global scope is the equivalent of a conventional smart contract. Everything defined in the global scope has an instance on all blockchain nodes and the instances are consistent globally. The shard scope defines a state that has one instance on each blockchain node of the shard and the instances are consistent in the shard.

Here is an example of state variables defined in different scopes:

contract Example { uint @address balance; // Has one instance for each possible value of type address. uint @global totalSupply; // Has only a single instance address @shard shardOperator; // Has one instance in each shard string @bool booleanName; // Has only two instances, one for true and one for false }

Scope Context

In multi-chain systems, states are distributed across shards and not accessible all at once when executing a transaction. In our model, a contract function is executed in a scope context

. Access to state variables and other functions is limited to this context.

Similar to state variable declaration, an optional scope class

is also added to function definition:

function transfer(address _to, uint256 _value) @address public returns (bool success)

The scope class

of a function defines the state variables it has access to. Only the subset of state variables defined with the same scope class

is visible to it. These variable might have multiple instances. During execution, the function can only see one instance of them. This instance is decided by msg.scope

, a new transaction property. It’s type is the same as the scope class

of the function where it is referenced.

Take the following code as an example:

contract ERC20 is IERC20 { uint @address public balance; function transfer(address recipient, uint amount) @address external returns (bool) { require(msg.sender == msg.scope); balance -= amount; // ... emit Transfer(msg.scope, recipient, amount); return true; } }

The function transfer

is defined with scope class of address

, therefore it has access to the state variable also defined as address

, i.e. balance

. Although balance

can have multiple instances (one for each possible address), they are distributed across the system in possibly multiple shards. When executing transfer

, only one instance is guaranteed to be available. Thus it is referenced directly using the identifier balance

, instead of indirectly indexed by msg.scope

. Access to other instances are only possible via asynchronous functional relays. We’ll discuss that later.

In addition to state variables, a function can also call other functions (either in the same contract or another contract) that are defined on the same scope class. Existing calling restriction are untouched and still apply (e.g. calling a non-pure function from a pure function is not allowed)

Accessing Global and Shard Scopes

Besides functions and state variables defined on the same scope class, a function also has access to the global

and shard

scopes:

- shard

scope has one instance in each shard, a contract function has access to the instance in the current shard where the contract is executed. It is not accessible from functions defined in the global

scope.

- global

scope has one instance globally and can be read by functions in any scope. To avoid conflicts, write access is only allowed inside @global

functions.

Similarly, @shard

functions can be called by any functions other than @global

functions; @global

functions that are defined as view

or pure

can be called by any other function, while other @global

functions that are not defined as view

or pure

are only callable among themselves.

The following table summarized accessibility of states / functions from functions in different scopes:

Function Scope

Access Global

Access Shard

Access Other

Global

Read & Write

Shard

Read Only

Read & Write(Current shard)

Other

Read Only

Read & Write(Current shard)

Read & Write(Current scope)

Other States of a Contract

Scopes only partition the storage of a contract, some other states of a contract, namely balance

, nonce

and code

, are not partitioned and is regarded as part of the global state. Any operation that might potentially modify these states are only allowed inside a @global

function. These include:

1. Creating another contract.
2. Using selfdestruct

.

1. Sending Ether via calls.
2. Using low-level calls.
3. Using inline assembly that contains certain opcodes.
4. The function is defined as payable

.

The above list basically duplicates the restriction solidity has on view functions, excluding those that modify the storage or emit events.

Emitted events are stored in log

and also considered as part of the contract state. We partition log

also into shards and each log item is stored in the shard that emits it. In this way, event emission is not limited to @global

functions but allowed in any function not defined as view

or pure

.

The reserved functions of a solidity contract: constructor()

, receive()

and fallback()

must all be defined as @global

since they might modify the contract state.

Asynchronous Functional Relay

During execution, a function only has access to a limited set of states in the current scope. Most behaviors of a contract involve operating on states in multiple scopes, e.g. even in a simple payment transfer function, it needs to first subtract the amount from the balance in the sender's scope and add it in the balance recipient's scope.

Unlike in single-box chain blockchain systems, in multi-chain systems the states of different scopes might be stored in different shards and not available on the same network node. Therefore, the execution must be relayed

from one scope to another asynchronously.

We introduce a new relay statement

to solidity grammar. A relay statement is much like a function call, proceeded by the relay

keyword:

relay functionName{options...}(arguments...);

A relay statement could also call functions defined in other contracts by proceeding the function name with a contract type / an address, like in regular external function calls.

A relay call is asynchronous. It packs all the options and arguments are in a transaction structure to be sent to the shard where the target scope is. Then it returns immediately without a return value.

Relay Call Options

A relay call is to pass the execution to another scope, it need specify the target scope. We add a new option scope

to call options, its type must be the same as the scope class of the called function. scope

could be omitted if the function being called is @global

or @shard

.

Relay calls are asynchronous and cannot automatically carry the current remaining gas like a regular external function call. Hence the relay call needs to explicitly specify the gas

option. The contract developer must take special care here to make sure that there's enough gas carried to complete the execution of the relay. On the other hand, since this carried gas will not be returned to the current execution environment, the developer should also leave enough gas behind for the current transaction to finish execution.

Storage Layout of State Variables

For @global

and @shard

state variables, their layout in the storage is unchanged. State variables defined with other scope classes can have multiple instances and each must be stored separately.

To achieve that, we regard such state variables as mappings, more specifically:

type @scopeclass identifier;

is regarded as:

mapping(scopeclass => type) identifier;

Since scopeclass

is limited to elementary type

, it's always allowed to use it as a mapping key type

. When reading / writing to the state variable, it's location in storage is calculated the same way as a mapping, with msg.scope

as the key. The nature of mapping guarantees that identifier

of different scopes are stored in different locations in the storage.

Gas Fee and balance

of Externally Owned Addresses (EOA)

Since multiple transactions are executed in parallel in a multi-chain system, the gas limit offer in a transaction signed by an EOA must be deducted before execution to avoid race conditions. This requires each transaction start from the shard where balance

of the EOA is modifiable. If balance of EOAs are stored like contracts' in the global shard, all EOA-created transactions would cause a congestion in the global shard, greatly reduce parallelity.

Therefore, we move the balance

of each EOA to the shard where that EOA belongs to. A transaction signed by an EOA starts its execution in the shard of that EOA by deducting the offered gas limit from the EOA's balance and then relayed to the target scope, carrying the gas limit with it. Any relay transaction are automatically relayed back to tx.origin

's scope after execution to return the remaining gas, if there's any. Since EOAs are expected to be relatively evenly distributed over the system, it is expected not to create any hot-spots with this design.

New EVM Opcodes

SCOPESIZE

and SCOPECOPY

msg.scope

is a sequence of bytes whose size is dependent on the function being called. Therefore, we add two new opcodes SCOPESIZE(0x49)

and SCOPECOPY(0x4A)

with format and gas fee similar to CALLDATASIZE

and CALLDATACOPY

:

OpCode

Name

Gas

Initial Stack

Resulting Stack

Mem / Storage

Notes

0x49

SCOPESIZE

2

.

len(msg.scope)

length of msg scope, in bytes

0x4A

SCOPECOPY

A3

dstOst, ost, len

.

mem[dstOst:dstOst+len-1]:=msg.scope[ost:ost+len-1]

copy msg scope

Unlike msg.data

, msg.scope

can have only one variable (the scope) in it, hence there’s no need to provide another opcode like CALLDATALOAD

to load individual words at specific indices.

RELAYCALL

We also add a new opcode RELAYCALL(0xFB)

for relays. It’s format and gas fee are similar to CALL

. It has two additional arguments, scopeOst

and scopeLen

, for the offset and length of the target scope in memory respectively. Besides that, a relay call does not have a return value and therefore does not take retOst

and retLen

as arguments.

OpCode

Name

Gas

Initial Stack

Resulting Stack

Mem / Storage

Notes

0xFB

RELAYCALL

AA-1

gas, addr, val, scopeOst, scopeLen, argOst, argLen

success

Sample Code

Here is how a contract that partially implements ERC20 may look like:

pragma solidity ^x.x.xx;

contract ERC20 { uint @global public totalSupply; uint @address public balance; mapping(address => uint) @address public allowance;

```
function _deposit(uint amount) @address internal {
    balance += amount;
}

function transfer(address recipient, uint amount) @address external returns (bool) {
    require(msg.sender == msg.scope);
    balance -= amount;
    relay _deposit{scope: recipient}(amount);
    emit Transfer(msg.scope, recipient, amount);
    return true;
}

function approve(address spender, uint amount) @address external returns (bool) {
    require(msg.sender == msg.scope);
    allowance[spender] = amount;
    emit Approval(msg.scope, spender, amount);
    return true;
}

function transferFrom(address sender, address recipient, uint amount) @address external returns (bool) {
    allowance[msg.sender] -= amount;
    balance -= amount;
    relay _deposit{scope: recipient}(amount);
    emit Transfer(msg.scope, recipient, amount);
    return true;
}

function _addtotalsupply(uint amount) @global internal {
    balance += amount;
}

function mint(uint amount) @address external {
    require(msg.sender == msg.scope);
```

```
    balance += amount;
    relay _addtotalsupply(amount);
    emit Transfer(address(0), msg.sender, amount);
}

function _subtotalsupply(uint amount) @global internal {
    balance -= amount;
}

function burn(uint amount) @address external {
    require(msg.sender == msg.scope);
    balance -= amount;
    relay _subtotalsupply(amount);
    emit Transfer(msg.sender, address(0), amount);
}

}
```

Rationale

Pack msg.scope

in msg.data

Instead of adding a new field .scope

to msg

, we also considered packing it inside msg.data

between function signature (the first 4 bytes) and the arguments. Note that this layout would be exactly as if there's a hidden first parameter of type of the scope class:

function transfer(address recipient, uint amount) @address; function transfer(address _scope, address recipient, uint amount); // msg.data for both of the above functions would be: // [Signature: 4 bytes][address: 32 bytes][address: 32 bytes] [uint: 32 bytes]

By packing this way, we could avoid introducing a new scope

field to the msg

structure and the opcodes SCOPESIZE

and SCOPECOPY

would be no longer needed. A function knows its scope class and can access it using CALLDATALOAD

to extract it from msg.data

.

However, since scope

defines the environment the function is executed in but not acts as part of the arguments passed into the function, we want to make this clear and leave it as a separate field.

Use a Reserved Contract Address for Relay Calls

Instead of introducing the new RELAYCALL

opcode, we also considered using a reserved contract address for relay calls. A solidity relay

statement would be compiled into an regular CALL

opcode and when the EVM forwards this to the host, the host would check the address and pack it as a relay transaction instead of synchronously calling the target contract. The reserved contract address could be a small integer like the Precompiled Contracts.

The downside of this is that the CALL

opcode's format is not identical to the proposed RELAYCALL

as it doesn't include scopeOst

/ scopeLen

in the arguments and retOst

/ retLen

is not needed in the case of relay. A workaround is to pack msg.scope

in msg.data

like proposed above, and always set retOst

/ retLen

to zero. A most serious issue is that it would make the CALL

opcode confusing because it can be either synchronous or asynchronous based on the address. In the end, we decide that adding a new RELAYCALL

opcode is more reasonable.

Transaction Atomicity and Sending Gas over Relay Transactions

Unlike regular calls, relay calls are asynchronous and if they fail, there's no automatic guarantee that the original transaction would be reverted because it already finished executed and included in a block on a possibly different chain. It's the developer's responsibility to take care of such cases and write code manually to revert the changes made by the original transaction by relaying back there.

A special case here is that a relay transaction fails due to not having enough gas to finish execution. In this case, it's also impossible to relay back due to lack of gas. Therefore, when writing a relay statement, the developer should plan ahead to send enough gas with the transaction to avoid such cases. Note that this is a common issue for cross-chain contract calls in any system and also exists for Rollups.

Move contracts' balance

outside global

Scope

Since balance

is considered part of the contract's global

state that is not partitioned, any operation that might modify it are limited to global

functions, including any call that sends value

. This might limit the flexibility of contracts.

A possible solution is to move a contract's balance

also to the shard where the contract's address, like EOAs. This requires any call that sends a value to be only executable in this shard. However, this would require some extra semantic to specify a special scope for the contract's address (e.g. @this

). We leave this to future work.

Backwards Compatibility

This proposal adds scope

to solidity grammar for state variables and functions. When not specified, it defaults to @global

and all old solidity contracts would live in the global

scope and their behavior would be unmodified.

It also adds a couple new opcodes to EVM but does not modify the behavior of any existing opcode, and thus is backward compatible for old contracts that don't use the new opcode or called via the new opcode.

The balance

of EOAs are moved to addresses, hence any contract that relies on getting the balance of an EOA would not work anymore.