PREDA: A Programming Model to Scale out Smart Contracts

PREDA DEV TEAM

Transactions of a single smart contract can only be processed within a single sequential execution engine (e.g. EVM) in widely adopted blockchains like Ethereum as well as in the state-of-the-art multi-chain systems like Polkadot and NEAR. Since few smart contracts contribute majority of transactions, a solution to scale out a single smart contract is crucial.

In this paper, we propose PREDA, a novel programming model to scale out any single smart contract by partitioning the contract state and dividing the transaction traffic, which are jointed handled by multiple independent execution engines that can be distributed and parallelized. Since the execution flow of a transaction may depend on contract states distributed on different engines, the key design of our approach is to decouple the transaction logic and the contract state in a scalable and efficient way by moving execution flow around instead of moving data between engines.

We implemented PREDA model by extending the existing Solidity language, which demonstrates that popular smart contracts with different level of complexity can be rewritten to gain scalability without taking care of the details of the underlying distributed system. In our experiments, PREDA model achieves significant performance and scalability advantages, and also exhibits promising expressiveness for general smart contracts.

1 INTRODUCTION

Since the emerging of Bitcoin [35], improving the throughput and scalability of blockchain has been a hot topic in both academia and industry. Among the existing works that address blockchain performance bottlenecks, sharding is an efficient method that divides the entire blockchain network into multiple shards and processes different transactions in the shards individually and simultaneously. Many sharding blockchains[9, 28, 32, 51–53, 55, 58, 59] are proposed in recent years, in which the throughput of payment transaction execution can be increased to hundreds of thousands of transactions per second (TPS), from 7 TPS of Bitcoin.

On the other hand, Ethereum [15] introduced a general but serial programming model, i.e., smart contract, expanding the scope of applications on the blockchain from simple payment to any custom programs. Nowadays, most blockchains support smart contracts. Some blockchains define their own smart contract programming languages, e.g., Solidity from Ethereum [49], Move from Facebook Diem [14], and Cadence from Flow [19]; some [9, 11] extend a general-purpose programming language like Rust and JavaScript; and also [43] provides an intermediate-level language that can be used in high-level languages like Solidity. In general, a smart contract is a collection of states as contract variables and program behavior as contract functions. After a smart contract is compiled and deployed on the blockchain, all participating nodes in the network have the compiled code replicated. When a transaction, an external input indicating a specific call to a contract function with arguments, is submitted to the blockchain, it is executed and validated by all nodes individually. With a large number of transactions, all nodes must process these workloads identically and in a consistent order. The contract states updated on these nodes are exactly the same as well. The blockchain system is thus essentially equivalent to a single state machine.

Performance of single-chain blockchain systems like Ethereum is extremely restricted as all transactions of all smart contracts are processed by a single instance of the execution engine, e.g. Ethereum Virtual Machine (EVM). Laterally, blockchain is scaled out by multi-chain blockchain systems [52, 56], which run one independent execution engine on each chain and process all transactions of a smart contract in one designated instance of the execution engine. Multi-chain systems work well when there are many smart contracts but each has a few transactions. However, from the perspective of a single smart contract, it gains no scalability in these multi-chain systems since only one instance of the execution engine can be leveraged. We observed transaction traffic of different smart contracts varies greatly, and on Ethereum, top-10 smart contracts by number of transactions contributed 26.98% of the total transaction volume in Q1 2022 [10]. These top smart contracts should be scaled out, but are not yet supported by existing methods.

Author's address: PREDA Dev Team, https://www.preda-lang.org/.

1.1 Smart Contract Scalability

Smart contract scalability is defined as the continuous improvement of transaction throughput and state capacity for a single smart contract when increasing the number of independent execution engines. To achieve this goal, we propose the PREDA programming model, which describes a smart contract in a way that can be distributed, parallelized, and scaled out by the underlying system using multiple execution engines.

With the PREDA model, transactions of a contract are divided and distributed for processing in different instances of execution engines without duplication. The states of the contract are partitioned and distributed without overlapping. In the ideal case, this approach allows for a linear scaling of overall transaction throughput and state capacity as the number of execution engines increases.

The key challenge in PREDA model is efficiently handling the dependency of the execution logic (code) and the contract state (data) while allowing execution engines to work independently and avoiding synchronization. The complete execution logic of a transaction may access multiple parts of the contract states, which may reside in different execution engine after state partitioning.

1.2 Distributed by Relay-Execution

When the execution of a smart contract function reaches a point that requires access to contract state residing on another execution engine, the execution flow is stalled until the required data is available. It is straightforward to move the required data from another execution engine so that the execution can be continued [31, 38], however it may introduce significant overhead of data transfer and complicated distributed locking for safe data modification. Furthermore, as for blockchain system, moving data from untrusted remote peers requires security proof to prevent data inconsistency and tampering, which is costly and inefficient as the required data is mutable and can vary in granularity.

We propose to move the execution flow of a transaction around while keeping partitioned contract data residing in their designated instance of execution engine at all times. Figure 1 illustrates the execution of a transaction function that requiring contract data resided in two different parts of the contract state. In the single execution engine system like EVM, as shown in Figure 1(a), the execution can be completed in one go but cannot be scaled. In a system with multiple independent execution engines, as shown in Figure 1(b), the execution of a smart contract function continue as long as the required data is available in the first execution engine (x). When an external data dependency is encountered (data is unavailable), a relay message will be emitted to initiate the subsequent execution in the second execution engine (y) at a later time.

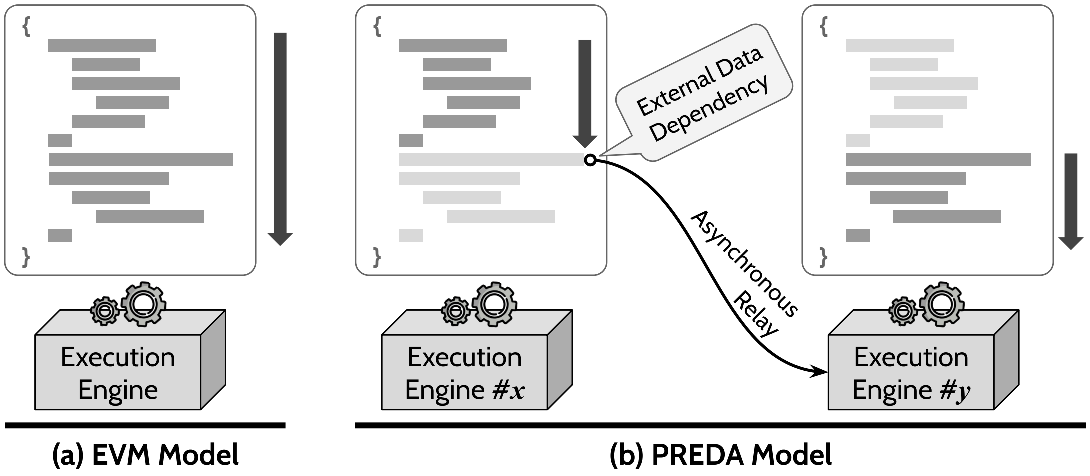


Fig. 1. Relay-Execution to facilitate the distributed execution of a transaction when depended contract data resided in different execution engines.

This Relay-Execution approach requires no moving or locking of contract states. Every partition of contract states is updated exclusively by its designated execution engines. The relay message carries the location of the interrupted point of the transaction function and a serialized package of local context, such as temporary variables, which is typically much smaller and more flexible for optimization. Verifying the integrity a relay against tempering is a typical built-in capability in most sharding blockchain and parachain systems. The Relay-Execution approach assumes that the code for all smart contracts is deployed in all execution engines, and a smart contract function can be executed in any instance of the execution engine at any time while working with different partition of contract state though. Since smart contract code is a small and constant dataset, fully replicating the code in all engines is a straightforward task.

1.3 Contribution

In this paper, we propose **P**arallel **R**elay-**E**xecution **D**istributed **A**rchitecture (PREDA), a novel programming model for scaling out smart contracts on sharding blockchains, parachain systems and layer-2 blockchains. PREDA model introduces

- **Programmable Contract Scopes** to define the partitioning of contract state based on the data access pattern of the application, which narrows the range of data access and minimize the data dependency.
- **Asynchronous Functional Relay** to describe the transaction logic with implicit data dependency exposed so that the execution can be easily moved across multiple execution engines.

The proposed programming model leverages existing consensus algorithm and transaction replication mechanism. No elements are introduced that compromise the security and decentralization of the blockchain system.

We have implemented the PREDA model as an extended Solidity language, incorporating additional syntax for programmable contract scopes and statements for asynchronous functional relay. We have developed a multi-thread parallel transaction processor on a single machine and a simplified sharding blockchain system distributed over the Internet, for the evaluation of the PREDA model.

We utilize the extended Solidity language to rewrite four smart contracts originally developed on Ethereum: Payment, Voting, AirDrop, CryptoKitties, and Million-Pixel. Subsequently, we conduct a series of experiments to compare them with their original counterparts on Ethereum. We primarily assess the performance of smart contract executions on a single machine, excluding the overhead associated with running the consensus protocol and network propagation. In a global testbed comprising 128 cloud virtual machines, we compare end-to-end performance. The results demonstrate that our work achieve promising scalability with an 256 shard configuration.

2 BACKGROUND

In this section, we provide the necessary background of this work, including the details of execution engines on blockchain systems and smart contract executions on Ethereum.

2.1 Multi-chain Systems

Multi-chain blockchain systems maintain multiple instances of chain of blocks in the network. Each instance has its own execution engine and an independent process of chain-forming and transaction replication. Multi-chain blockchain systems can be categorized based on various metrics. Figure 2 shows two typical structures of a multi-chain system: the sharding blockchain, and the parachain system, based on the implementation methods of cross-chain invocation. When there is a cross-chain invocation, parachain systems [13, 56], as shown in Figure 2(b), use a dedicated relay chain to forward the relay from one parachain to another parachain; while sharding systems, as shown in Figure 2(a), allow any participating node in a shard [52, 53, 55, 58] or the end-user initiating the original transaction [9, 28] to send the relay directly through the underlying P2P network. Additionally, multi-chain blockchain systems can be further divided based on how smart contracts are deployed and executed on shards or parachains. Some systems [9, 13, 28, 55] allow a smart contract to be deployed and executed on all shards or parachains, while others [52, 56, 58] confine a smart contract to a specific shard or parachain. In the former scenario, cross-chain invocations can occur within or across contracts;

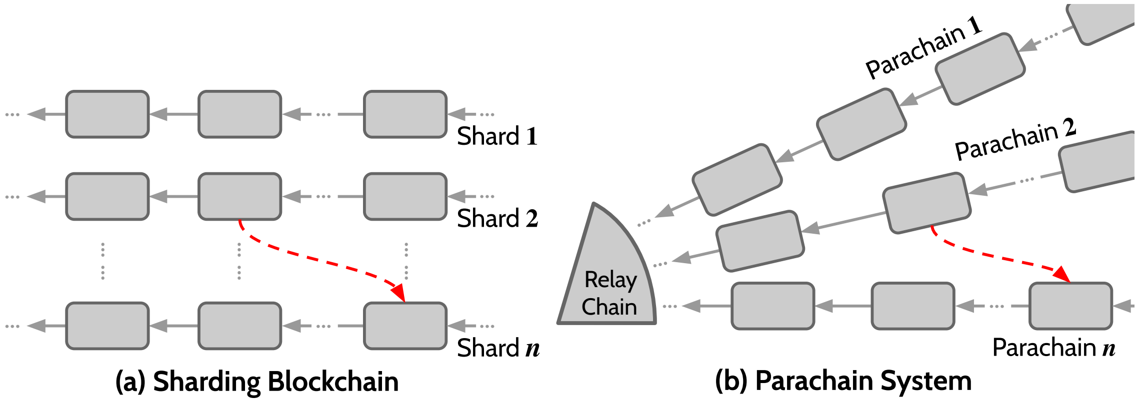
whereas in the latter, cross-chain invocations are limited to across contracts only. One of the primary objectives of the PREDA model is to establish a general-purpose and scalable framework for multi-chain blockchain systems, irrespective of low-level implementation details such as relay mechanisms, deployment strategies and execution methods.

Consensus algorithms for multi-chain systems share a common capability in addition to those for single-chain systems like Ethereum. For example, to make a cross-contract invocation between two contracts deployed in different chains, a *relay transaction* will be composed in the caller's chain, then be forwarded and inserted into the callee's chain. A multi-chain consensus algorithm provides an efficient way to verify the authenticity of an inbound relay transaction without accessing information from the caller's chain, which are illustrated by red arrowed lines in figure 2. The proposed PREDA model reuses this capability to make asynchronous functional relay, which securely moves the execution flow a transaction to a target execution engine where the required contract state resides.

Disregarding the consensus details, an execution engine on a blockchain system can be abstracted as a sequential state machine as illustrated in figure 3. With a sequential manner, it takes transactions from an ordered queue, executes contract function as each transaction indicates and updates the involved parts of contract state accordingly. The ordered transactions are organized batch-wise as blocks which are composed by the block creator in consensus layer. Any external input is received as a transaction, unordered, which can be user-signed normal transactions or verified relay transactions from other execution engines. These external transactions are transferred over a broadcast network and cached until being inserted into the ordered queue for execution, a.k.a. *memory pool*. When executing a transaction on a multi-chain system, outgoing *relay transactions* might be emitted and then passed from the initiating chain to a destination chain, where relay transactions are pooled, confirmed and finally executed.

2.2 Smart Contracts

Smart contracts broaden the application of blockchain, supporting from pure payment applications to arbitrary customized applications. Figure 4(a) shows an example of a simplified ERC20 contract written in Solidity. The code snippet contains a contract state, i.e., `balances` representing the balances of the corresponding addresses, and a contract function `transfer`, which is to transfer a number of `amount` tokens from the transaction sender `msg.sender` to a `payee`. In Ethereum, once a contract is successfully deployed, each node has the compiled contract and stores the bytecode for future execution in its local Ethereum Virtual Machine (EVM) [50]. The states and blocks are stored in a key-value store, e.g. LevelDB. When a user submits a transaction to invoke the function `transfer` with corresponding parameters `payee` and `amount`, miners first validate the transaction, e.g., if it has a valid signature, and then execute the function in the EVMs. In this case, the opcodes, e.g., `SLOAD`, `SUB`, `ADD`, and `SSTORE`, are used. The state of the sender, i.e., `balances[msg.sender]`, is updated by withdrawing a number of `amount` tokens with the opcode `SUB`, and the state of the receiver, i.e., `balances[payee]`,



(a) Sharding Blockchain
(b) Parachain System

Fig. 2. Multi-chains systems have multiple blockchains working cooperatively in parallel, on each an instance of execution engine is running.

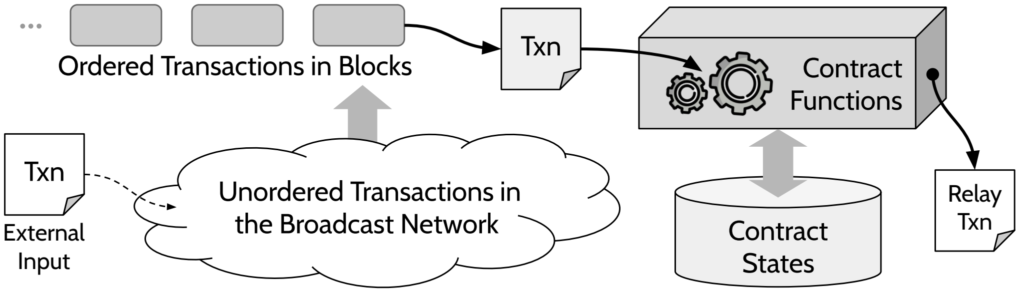
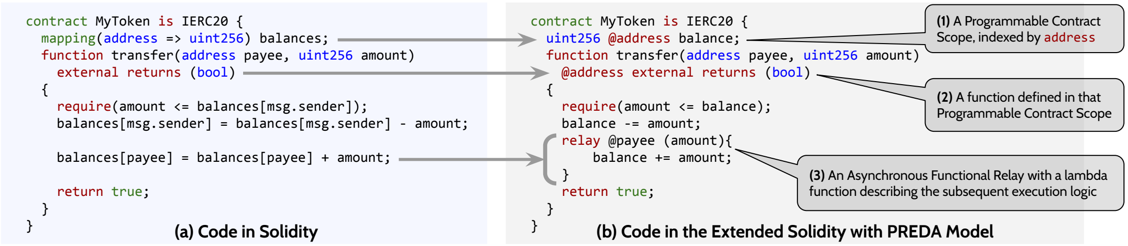


Fig. 3. A unified abstraction of an execution engine on blockchain systems.



```
contract MyToken is IERC20 {
  mapping(address => uint256) balances;
  function transfer(address payee, uint256 amount)
    external returns (bool)
  {
    require(amount <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - amount;

    balances[payee] = balances[payee] + amount;

    return true;
  }
}
```

(a) Code in Solidity

```
contract MyToken is IERC20 {
  uint256 @address balance;
  function transfer(address payee, uint256 amount)
    @address external returns (bool)
  {
    require(amount <= balance);
    balance -= amount;
    relay @payee (amount){
        balance += amount;
    }
    return true;
  }
}
```

(b) Code in the Extended Solidity with PREDA Model

(1) A Programmable Contract Scope, indexed by address

(2) A function defined in that Programmable Contract Scope

(3) An Asynchronous Functional Relay with a lambda function describing the subsequent execution logic

Fig. 4. A glace of a smart contract with the proposed PREDA model.

is updated by depositing a number of `amount` tokens with the opcode `ADD`. Smart contracts in Ethereum are executed instruction by instruction and transaction by transaction. After the execution, a miner selected by the consensus protocol inserts the executed transactions into a block and sends it to the network. After receiving a block, a full node executes its transactions and updates its local states accordingly.

Existing studies, e.g., [12, 16, 20, 24, 41, 42], allow multi-threaded execution of smart contract transactions. However, these methods are not scalable since each node needs to execute all transactions and store all states. On multi-chain blockchain systems, payment transactions can be executed by multiple shards or parachains in parallel. However, for smart contract transactions that invoke arbitrary user-defined functions, existing systems either use a dedicated chain to execute smart contract transactions [51, 53] or do not support smart contract transactions so far[28, 55, 59].

3 PREDA PROGRAMMING MODEL

In Solidity on Ethereum, a smart contract is defined as a set of variables (contract state) and functions (transaction functions) that update variables. As illustrated in figure 4(a), both state variables and functions are defined in the global scope of the smart contract. However, this global scope presents two significant challenges that hinder the effective and efficient scaling of smart contracts.

First and formost, efficiently partitioning the contract states requires an understanding of the data access patterns associated with state variables. While static code analysis can help identify the boundaries of the contract state with precision, an optimal design necessitates a deeper understanding of the application's nature being developed, enabling the partitioning of state variables based on how they are utilized. We introduce **Programmable Contract Scope (κ-scope)**, a solution that enhances expressiveness in describing contract state partitioning and provides the flexibility needed to enhance scalability and optimization.

Secondly, a function defined in the global scope necessitates the availability of the entire contract state for execution, as its data dependencies can span across arbitrary portions of the contract state. This requirement is impractical in the context of building a scalable system. To address this challenge, we propose a solution wherein the scope within which a function operates is narrowed down, ensuring that its data dependencies are predetermined irrespective of the actual values of invocation arguments. **Asynchronous Functional Relay (λ-relay)** is introduced to decompose the execution of a transaction to multiple invocations of these scope-narrowed functions in the order of data dependency, asynchronously across multiple independent execution engines.

3.1 Semantics

A programmable contract scope ϕ is defined as a collection of variables $\{\mathscr{S}\}$ and functions $\{\mathscr{F}\}$ that are restricted to access only variables within the same scope. In a smart contract, there can be a great number of programmable contract scopes. These are indexed by a key k with a built-in type like string or integer. A set of keyed κ-scopes Φ can be formulated as

$$\Phi : \quad \phi_k \Rightarrow \langle \mathscr{S}, \mathscr{F} \rangle, \quad k \in \mathscr{K} \tag{1}$$

, in which $\mathscr{K}$ denotes the set of all possible values of k.

A function $f_{\phi_k} \in \mathscr{F}$ of a κ-scope ϕ_k has immediate access to all variables $\mathscr{S}$ and functions $\mathscr{F}$ only within that κ-scope besides its invocation arguments and the execution context (e.g. block height, message sender and etc). Unlike functions in Solidity, a function in the PREDA model is invoked by providing the current κ-scope (**target κ-scope**) to start, an analogy to *this* pointer in C++. To continue the execution flow dealing with state in another κ-scope $\phi_{k'}$, an asynchronous invocation of a function $g_{\phi_{k'}}$ will be initiated, which is formulated as a λ-relay :

$$\left\langle \phi_{k'}, g_{\phi_{k'}}, R \right\rangle \tag{2}$$

, in which $\phi_{k'}$ is the target κ-scope and R is the vector of the invocation arguments that provided by the caller f_{ϕ_k}.

Figure 4(b) shows the PREDA version of the simplified ERC20 contract that can be scaled out. In part (1), a set of κ-scopes keyed by **address** type is defined to represent users' balance, which is equivalent to the map definition in Solidity in the same line of Figure 4(a) but describes a set of fine-grained separable states for partitioning. All κ-scopes in this example has the same definition of variables but each has a unique instance. Accordingly, the transfer function is defined in the same set of κ-scopes in part (2), which is invoked by providing k with the payer's address as the target κ-scope. In part (3), to proceed with the deposit to the payee k' after a successful withdraw, a λ-relay is initiated with $\phi_{k'}$ as the target κ-scope, which adds funds to payee's balance and is executed by an engine that hosts the state of $\phi_{k'}$.

In a smart contract, there can be multiple κ-scopes having variables and functions defined. Multiple functions and variables of arbitrary types including containers can be defined in a κ-scope. Multiple λ-relays can be initiated in a single function call, conditionally or unconditionally. λ-relay initiation can be recursive which allows a transaction execution flow being moved multi-hops across different instances of the execution engine. Examples dealing with more complicated logic of transactions are discussed in section 6.

Special κ-Scopes

Engine Scopes: One instance of κ-scope $\phi_{\theta i}$ is built-in for each instance of execution engine to represent a scope that is available for immediate read/write by any function executing in the current execution engine.
Global Scope: A built-in κ-scope ϕ_Ω that is logically singleton in the entire network. Its states are updated consistently across all execution engines in a multi-chain system, which provides a scope that is available for immediate read access by any function in the network.

Besides the special rules of the data availability described above, both $\phi_{\theta i}$ and ϕ_Ω have the same way for defining variables and functions, the same restriction for cross-scope data access and the same requirement of initiating λ-relays.

3.2 Partitioning

To decouple the smart contract implementation with the underlying multi-chain architecture, the PREDA model strictly prohibits referring a specific instance of the execution engine or making assumption of the underlying distributing configuration of multiple execution engines (e.g. total # of engines). This also frees the developer from dealing with details of the underlying distributed systems.

In the PREDA model, programmable contract scopes described in a smart contract expose fine-grained boundaries of contract states that can be partitioned, and leave the actual partitioning strategy to the host of execution engines on following considerations:

- A partition scheme should evenly partition the entire value space of k without overlapping, which uniquely maps a k to an instance of the execution engine in the network.
- Partition mapping should be resolved only based on the k of a κ-scope ϕ_k and identifies a single instance of the execution engine without ambiguity.
- Contract states in storage are indexed by k, and are written according to the current κ-scope ϕ_k.
- On initiating a λ-relay, the host should convert the relay whose target κ-scope is hosted by the current execution engine into a local invocation instead of composing and emitting a relay transaction.

For example on a sharding blockchain with 2^n shards, the partition mapping can simply be the first n-bits of the crc32k. Each execution engine owns a unique instance of states in the engine scope $\phi_{\theta i}$ for i-th execution engine, and maintains a copy of states in the global scope ϕ_Ω, which is consistent across all instances of the execution engines. An engine scope $\phi_{\theta i}$ is not allowed to be the target of a λ-relay nor being referred by specifying θi. Variables and functions only in the $\phi_{\theta i}$ can be accessed by the i-th execution engine, which is referred implicitly as part of the current execution context.

3.3 Relaying

A function executing in a κ-scope ϕ_k is required to initiate a λ-relay to proceed execution that deals with state variables in another κ-scope $\phi_{k'}$. Figure 5 illustrates the workflow of the λ-relay in the example shown in Figure 4, a successful execution of `transfer` transaction will emit a λ-relay, which will be converted to a relay transaction by the host of the execution engine. The relay transaction will be passed to the memory pool as an unordered pending transaction in the destination execution engine, and later confirmed and executed there. The actual mechanism of passing a λ-relay to the destination execution engine is not defined in the PREDA model, but handled by the host of the execution engines and the underlying multi-chain system. It is required that the host and the multi-chain system have the following capabilities:

- A λ-relay can be converted to a relay transaction with target κ-scope, identifier of the function to be invoked, and arguments if any.
- A block should carry a proof (e.g. Merkle root) of the complete set of all emitted λ-relays so that the integrity of all outgoing relay transactions can be verified by other nodes.
- A relay transaction should carry a proof (e.g. a Merkle path) for verifying that it is emitted by a transaction confirmed in a specific block from the initiative execution engine.
- A relay transaction can be transferred to the memory pool of the destination execution engine as an unordered pending transaction, and awaits being confirmed and executed.

A λ-relay to the global scope ϕ_Ω logically undergoes the same workflow as normal λ-relays. Since a global relay transaction is broadcast, duplicated and transferred to all execution engines, the states in the global scope will be consistent across all execution engines. At any block height, blocks in different execution engines have an ordered set of global relay transactions that is consistent across all engines. The global relay transactions will be executed before any transaction specific to a particular execution engine in that block. As discussed in section 3.1, a λ-relay to an engine scope is not allowed.

Cross-Contract Invocations can be carried out immediately without requiring a λ-relay in the PREDA model, as long as the invocation is within the same κ-scope. A cross-contract invocation across different κ-scopes is required to initiate a λ-relay targeting the callee's κ-scope. Thus, taking cross-contract invocations into account, the λ-relay definition in Equation 2 is extended to

$$\langle c', \phi_{k'}, g_{\phi_{k'}}, R \rangle . \tag{3}$$

c' denotes the smart contract whose function is invoked.

3.4 Parallelism

Transactions dealing with different κ-scopes are executed inherently in parallel since those are separately processed by parallel instances of independent execution engines. The PREDA model ensures that execution of any transaction has access restricted to contract state within the current engine, which allows execution engines to be driven by different threads on a multi-core computer without worrying about the thread safeness of data access, and to be deployed on different computers without resource sharing.

The relay-execution approach described in section 3.3 actually decomposes a complete transaction into multiple **Micro-Transactions**, or μ-Txns, each reads or writes a limited set of κ-scopes that has guaranteed availability in a single execution engine. μ-Txns scattered in different execution engines are executed in parallel while ones in the same execution engine are processed sequentially, which avoids race conditions and needs of concurrency protection of data access. In the example shown in Figure 4, the transaction is decomposed to a `withdraw` step and a `deposit` step. As long as their target κ-scopes are mapped to different execution engines, the two μ-txns are processed in parallel, along with more μ-txns of the `transfer` transactions. Parallelization at a granularity of μ-txn provides much better scalability and load balancing than those at smart contract level as other multi-chain blockchain systems did [52, 56, 58].

4 CRYSTALITY: THE EXTENDED SOLIDITY

To realize the proposed programming model, we developed the **Crystality** language by extending the widely adopted smart contract language, Solidity, which is originally developed for Ethereum and EVM. A few syntaxes are introduced to the existing Solidity language for declaring variables and functions in κ-scopes and making λ-relay invocations. A

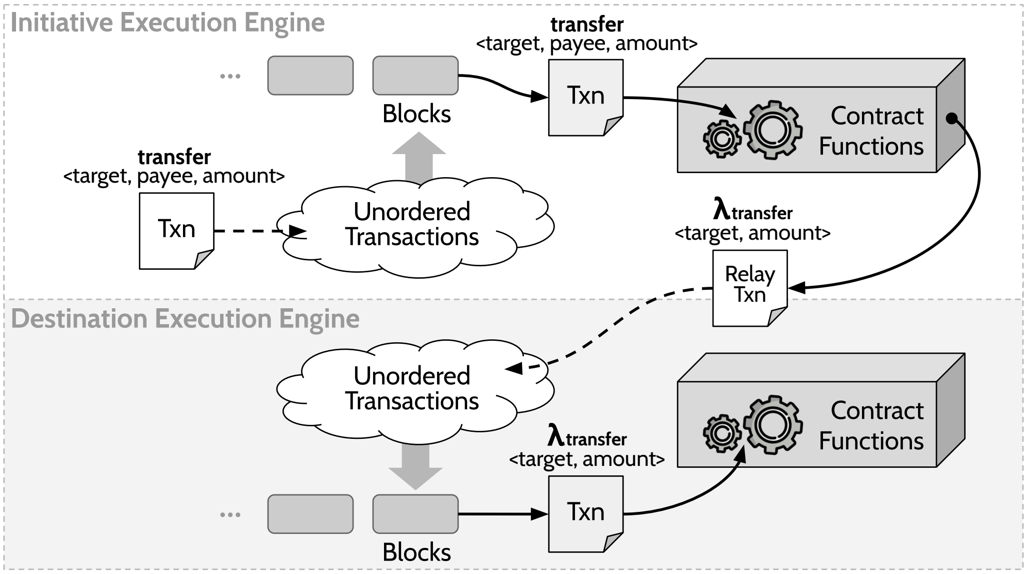


Fig. 5. Workflow of the λ-relay for `transfer` in the `MyToken` contract (figure 4).

transpiler is developed to convert the Crystality code to conventional Solidity code, by mapping new syntaxes to special raw invocations for new behaviors and checking scope compatibilities for error detection.

4.1 Variables in κ-Scopes

In Crystality, a state variable is defined and instantiated in a κ-scope, or for each key of keyed κ-scopes as:

```
var_type @scope var_name;
```

where `@scope` specifies a κ-scope which can be the global scope `@global`, the engine scope `@engine`, or when `scope` is a name of Solidity elementary typename to specify keyed κ-scopes like `@address` or `@uint`. `@global` can be omitted which is the default κ-scope specifier.

A variable definition with `@global` or `@engine` will be converted simply as

```
var_type var_name;
```

in Solidity and the scope specifier will be recorded in the symbol table in the transpiler runtime. Any reference to the variable will be converted to Solidity as is.

A variable definition with keyed κ-scopes will be converted to a mapping in Solidity:

```
mapping(scope => var_type) var_name;
```

with the scope specifier recorded in the symbol table for scope compatibility check when the variable is referred. A reference to the variable in a function will be converted to Solidity as a map access:

```
var_name[_target]
```

, in which `_target` is a built-in const value k representing the target scope ϕ_k.

4.2 Functions in κ-Scopes

A function is always declared in a κ-scope as

```
function func_name(arg_type arg, ...)
         @scope qualifiers returns (ret_type){ ... }
```

Similar to variable definition, `@scope` can be `@global`, `@engine` or a Solidity elementary typename. Again, `@global` is the default κ-scope specifier, which can be omitted. In cases of `@global` or `@engine`, the declaration will be simply converted to Solidity by removing `@scope` as

```
function func_name(arg_type arg, ...)
         qualifiers returns (ret_type){ ... }
```

with its scope specifier record in the symbol table.

When the `@scope` is a keyed κ-scope, the key k of the target scope ϕ_k is inserted as the first argument of the function like

```
function func_name(scope _target, arg_type arg, ...)
        qualifiers returns (ret_type){ ... }
```

The built-in constant `_target` is introduced as an argument to allow accessing variables defined in the current keyed κ-scope as described in section 4.1.

Code in a function body has immediate access restricted to variables and functions in the target κ-scope by referring corresponding symbols. While special κ-scopes such as `@global` and `@engine` have exceptions of the isolation rules for accessing variables and functions as mentioned in section 3.1, symbols of variables and functions defined in the special κ-scopes are merged with ones in the target κ-scope without scope qualifications. To this end, we require symbols defined in any κ-scope have unique names within a smart contract.

In the target κ-scope ϕ_k or in the current engine κ-scope $\phi_{\theta i}$, variables and constant functions defined in global scope are merged for read-only access. Symbols defined in the current engine κ-scope $\phi_{\theta i}$ are merged into the target κ-scope ϕ_k and allow full access of both reading and writing.

4.3 Relay to a Target κ-Scope

To continue execution logic involving contract states in a different κ-scope other than the target κ-scope without desired immediate access, a λ-relay invocation should be made as

```
relay @key (var1, var2, ...){ ... }
relay @global (var1, var2, ...){ ... }
```

, which defines a lambda function and emits a λ-relay with it. `relay` is a new keyword for making a λ-relay invocation and is followed by the target specifier which can only be the `@global`, a specific `key` of keyed κ-scopes or an expression resulting a key.

The `relay` invocation will be converted to Solidity code as an EVM message call on a magic contract address that can be recognized as a λ-relay invocation by the EVM host. So that such a λ-relay invocation can be captured and reinterpreted as a cross-scope relay transaction if necessary. The transpiler-converted Solidity code is shown as follows, which are equivalent to the Crystality code above:

```
address(_magic_address_kappa).call(
  abi.encodeWithSignature(
    "unique_funcname_k(scope,var_type1,var_type2,...)",
    key, var1, var2, ...
  )
);

address(_magic_address_global).call(
  abi.encodeWithSignature(
    "unique_funcname_g(var_type1, var_type2, ...)",
    var1, var2, ...
  )
);
```

`_magic_address_kappa` is a constant built-in address representing a λ-relay on a normal κ-scope ϕ_k, and `_magic_address_global` is for indicating the global scope ϕ_Ω. Such an invocation will be captured by the EVM host and converted to an outgoing relay transaction. When the relay transaction is received and confirmed, a private function in the target κ-scope will be invoked, which is transpiler-generated by taking the body of the the lambda function from the λ-relay invocation.

```
function unique_funcname_k
  (scope _target, var_type1 var1, var_type2 var2, ...)
  @scope private { ... }

function unique_funcname_g
  (var_type1 var1, var_type2 var2, ...)
  @global private { ... }
```

`unique_funcname_*` are unique function names within a smart contract generated by the transpiler.

The transpiler-generated functions converted from a λ-relay is guaranteed to be invoked only from the function where it is defined. It can be a named function to be invoked in a λ-relay without a function body as

```
relay @key named_function_k(var1, var2, ...);
relay named_function_g(var1, var2, ...);
```

In these cases, functions being invoked should be defined as

```
function named_function_k(arg_type arg, ...)
  @scope public { ... }

function named_function_g(arg_type arg, ...)
  @global public { ... }
```

A λ-relay to a named function can also be cross-contract with the function name scoped by the name `ExtContr` of the contract like

```
interface ExtContr {
  function named_function_k(arg_type arg, ...)
    @scope public;

  function named_function_g(arg_type arg, ...)
    @global public;
}

relay @key ExtContr(c_addr).named_function_k(var, ...);
relay ExtContr(c_addr).named_function_g(var, ...);
```

, in which `c_addr` is the address of the smart contract `ExtContr` actual deployed.

4.4 Transpilation for EVM

We employ a two-stage process to compile a Crystality smart contract to EVM bytecodes. First, a transpiler is developed to convert Crystality code to conventional Solidity code. We uses ANTLR [1] to generate the parser code based on the Solidity syntax definitions with Crystality extensions. The resulting abstract syntax tree (AST) is walked through to generate the Solidity code with necessary conversion and auxiliary code generation described above. Second, the generated Solidity code is compiled using the widely used SOLC [5] compiler to generate bytecodes that can be executed on an unmodified EVM. As an example, the simplified `ERC20` smart contract in figure 4 will be transpiled as

```
contract MyToken is IERC20 {
  mapping(address => uint256) balance;
  function transfer
```

```
      (address _target, address payee, uint256 amount)
        external returns (bool)
    {
      require(amount <= balance[_target]);
      balance[_target] -= amount;
      address(_magic_address_kappa).call(
        abi.encodeWithSignature(
          "_lambda_transfer_0(address, uint256)",
          payee, amount
        )
      );

      return true
    }

    function _lambda_0_transfer
      (address _target, uint256 amount)
    {
      balance[_target] += amount;
    }
  }

```

We use EVMOne[22] as the EVM implementation in our system to execute bytecode generated by SOLC. EVMC[3] is the standard to communicate with EVMOne. We implemented its `HostContext` interface to provide the EVMOne with capabilities of accessing state storage, handling message call and exposing metadata of current block and transaction. In addition to those, all λ-relay invocations are captured by overriding the `HostContext::call` function of the EVMC interface using the magic address `_magic_address_kappa` for keyed κ-scopes ϕ_k and `_magic_address_global` for the global scope ϕ_Ω. The target scope, function and arguments, as encoded by `abi.encodeWithSignature`, are passed to the underlying multi-chain system for composing the relay transaction and forwarding to the execution engine that hosts the target κ-scope .

4.5 Writing Scalable Smart Contracts

Crystality enables that a smart contract can be written in a scalable way based on PREDA programming model. While it relies on developers to separate the contract states in different κ-scopes using programable contract scope syntax based on the nature of the business logic, and decompose the transaction workflow using asynchronous functional relay syntax accordingly. The design goal is to minimizes amount of the contract states hosted in global scope ϕ_Ω and minimizes the transaction traffic processed in the global scope. Crystality provides flexibilities for developers to tweak and optimize smart contracts so that the understanding of the actual patterns of workload in the runtime can be leveraged.

Note that, any Solidity code can be compiled in the proposed Crystality system as is, which is actual put everything in the global scope. Such a smart contract will not be scalable at all and is equivalent to running a smart contract on a single-chain blockchain system.

Automatically analyzing a Solidity smart contract, decomposing data dependencies and separating contract states based on static code analysis or profiling with exemplar transaction traffic is an interesting research topic. We leave this to future works, while we provide the target for such automatic conversion as a starting point.

5 SYSTEM DESIGN AND IMPLEMENTATIONS

To evaluate the Crystality language with the proposed PREDA programming model, we developed a smart contract execution module as described in Section 4.4. Two testbeds are developed for testing the smart contract execution module: one for running on a multi-core single machine and the other for a distributed scenario on open Internet.

The PREDA smart contract execution module uses Solc[5] 0.8.18 and EVMOne[22] 0.8.0 for Solidity compilation and bytecode execution. Different implementations of EVM host interfaces are developed for adapting the two testbeds.

5.1 On a Multi-Core Single Node

We evaluate the pure execution performance of our method on a multi-core single node. We call this testbed as Multi-threading Transaction Processor, which minimizes overheads that are unrelated to smart contract execution such as block composition, data communication, signature verification, and consensus algorithm.

Using n cores on the test computer, n 1 execution units are allocated. Each has an independent instance of the EVM engine, a lock-free queue [18] for pending transactions and a working thread that drives the processing. One of the execution unit $\mathbb{U}_{\Omega}$ is dedicated for transaction processing and state updating in the global scope ϕ_{Ω}, and the rest n execution units $\mathbb{U}_{\theta i}$ work for all n partitions of keyed κ-scopes.

In each execution unit, transaction processing is performed batch-wise. Each batch will process transactions as much as possible until exceeding a predefined total gas limit or the pending transaction queue being drained. The processing is an infinite loop which undergoes the following steps:

- Finish the batch in $\mathbb{U}_{\Omega}$ for the global scope, while all the rest of units $\mathbb{U}_{\theta i}$ await.
- Process the batch in all units $\mathbb{U}_{\theta i}$ in parallel, while the unit $\mathbb{U}_{\Omega}$ awaits. Since contract state in global scope are read-only to all $\mathbb{U}_{\theta i}$, there would be no race condition.
- All units $\mathbb{U}_{\theta i}$ are finished when the total gas exceeds the limit.

Note that the total gas is the sum of the gas consumptions across all $\mathbb{U}_{\theta i}$ units to ensure that utilization of CPUs are balanced and all units finished nearly at the same time. When processing in a batch, all relay transactions emitted are collected and dispatched to the pending queue of the corresponding unit on the end of the current batch. Pending relay transactions are processed prior to pending normal transactions.

5.2 On a Sharding Blockchain System

We also evaluate the end-to-end scalability of the proposed model on a sharding blockchain system deployed on open Internet. We developed a simplified homogenous sharding blockchain system similar to NEAR[52], which creates n blocks for n shards at each block height synchronously as shown in figure 10. In addition, a global chain is introduced to process transactions and update states in global scope ϕ_{Ω} only. Global chain provides a globally synchronized consistent view of global scope at every block height. Designing a sharding blockchain is out of the scope of this paper, we leave details in appendix A.

6 EVALUATION

6.1 Crystality Smart Contracts

Besides the ERC20 contract, as shown in Figure 4, we rewrite four other widely used smart contracts in Crystality and use them to evaluate the performance of the PREDA model. They are `Voting`, `AirDrop`, `CryptoKitties`, and `MillionPixel`. The corresponding code can be found in Appendix B. In the following experiments, we use the same software setups to run these smart contracts. For `MyToken`, we use the Etherscan API [4] to obtain historical WETH [8] Token transactions in Ethereum, from block height 4,719,568 (i.e., the creation time of WETH) to block height 18,184,075 (i.e., at 08:00 am, EST, September 21, 2023). The dataset includes more than 197.2 million transactions from 877,664 addresses. We re-execute the first 1,000,000 transfer transactions in our experiments. For the other smart contracts, we randomly generate 1,000,000 Ethereum addresses.All these addresses are used in `Voting` and `MillionPixel` contracts and 10% of them are used in `AirDrop` and `CryptoKitties` contracts to issue transactions, with one transaction issued per address. Details on the numbers of different types of transactions will be discussed later.

Table 1. Numbers of different types of Crystality Txns in 16 shards

	MyToken	Voting	AirDrop	CryptoKitties	Million-Pixel
origin_txns	1,000,000	1,000,001	100,000	100,001	1,000,000
total_relays	1,000,000	33	500,000	400,033	1,000,000
intra_relays	64,839	0	32,720	18,698	62,687
cross_relays	935,161	33	467,280	381,335	937,313
global_txns	0	17	0	17	0
shard_txns	2,000,000	1,000,017	600,000	500,017	2,000,000
total_txns	2,000,000	1,000,034	600,000	500,034	2,000,000

6.2 On Multiple Single Nodes

We first evaluate the Crystality smart contracts on two different multi-core nodes, respectively. The first node is a powerful server that has an AMD EPYC 7742 (64-core, 3.4GHz) CPU and 2TB memory, running on Linux Ubuntu 20.04. The second platform is a desktop machine that has an Intel i7-10700 (8-core w/hyper-threading, 2.9GHz) CPU and 32GB memory, running on Linux Ubuntu 23.10. The number of transactions is kept the same when the number of shards increases. Each thread has its affinity set to a dedicated CPU core.

Figure 6 shows the performance numbers in Transactions per Second (TPS) on the AMD/Linux machine for all five Crystality smart contracts. Overall, we can achieve 30.3x to 56.1x TPS improvement when using 64 shards over 1 shard. Performance numbers of the equivalent Solidity smart contracts are also included. Since EVM can only sequentially execute Solidity transactions and different EVMs execute the exactly same transactions in Ethereum, the overall TPS of running Solidity smart contracts will not improve even with increasing number of EVMs. As a result, we only draw the TPS of running Solidity smart contracts in 1 shard with "Eth-" as the prefix. Compared to Solidity, Crystality contracts can achieve 17x to 42.9x TPS when run on 64 shards.

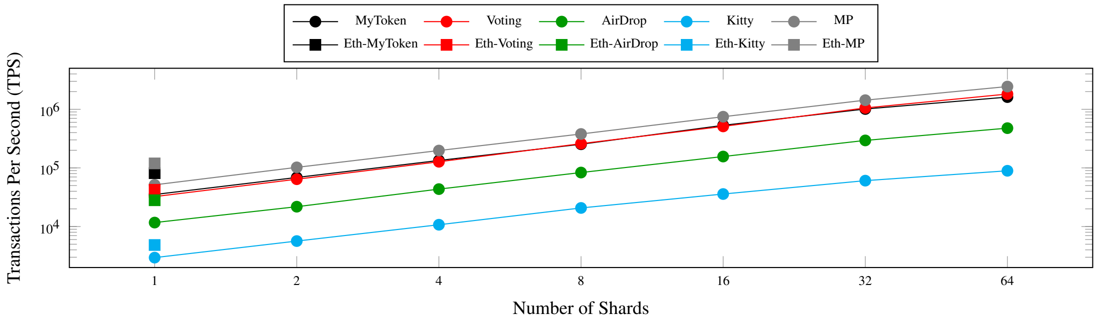
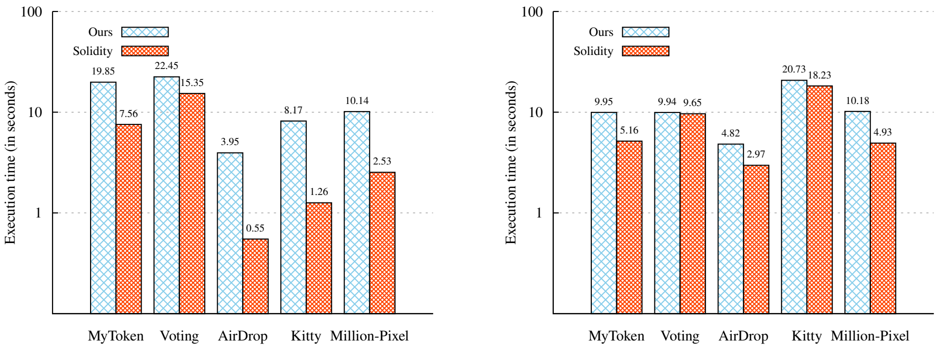


Fig. 6. Performance on 64-core AMD/Linux node

We collect the numbers of different types of Crystality Txns when using 16 shards to run the contracts and include them in Table 1. The "origin_txns" row represents the amount of transactions initiated from end-users. The "total_relays" row represents the amount of total relay transactions, which includes the intra-shard relays denoted as "intra_relays" and the cross-shard relays denoted as "cross_relay". The row "global_txns" and "shard_txns" represent the amount of transactions executed in execution unit $\mathbb{U}_\Omega$ and other $\mathbb{U}_{\theta i}$s respectively. The "total_txns" row means the total amount of transactions including both.

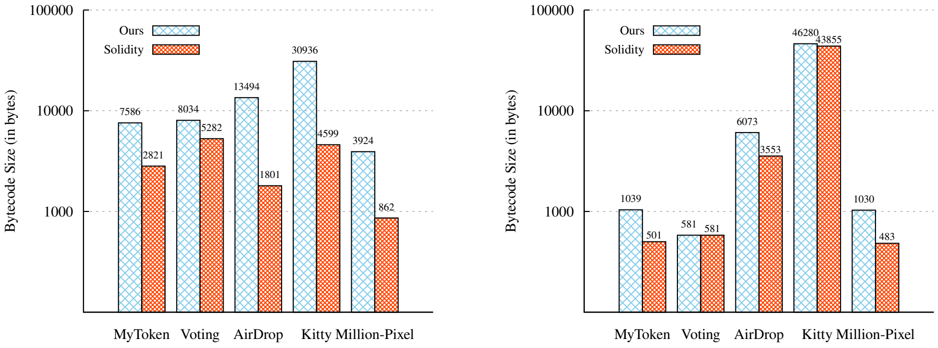
As best practice, programmers should (1) eliminate the global transactions as much as possible, because these transactions require global synchronization; and (2) try executing transactions in engine scope without emitting too many relays, because a relay leads to the switch of execution flow from one engine to another engine. As shown in the table, all five Crystality smart contracts issues very few or zero global transactions. For example, `Voting` contract executes 1 global transaction in its `finalize()` function to stop the voting, 16 cross-shard relays are emitted from the global to shards to request partial voting results, and 16 global transactions, each of which accumulates a partial voting result from a shard when using 16 shards. As a result, Crystality `Voting` contract scales the best with the number of shards, which is 56.1x TPS improvement with 64 shards.

When run on one single shard, the TPS of a Crystality contract is typically lower than its Solidity equivalent. In order to interpret the performance numbers, we look into the details of our transpiler-generated Solidity code shown in Listing 4.4. The major difference compared to the Solidity equivalent (as shown in Figure 4(b)) is that our transpiler-generated code implements the relay semantic with the `address.call()` method, which generates a relay transaction in the EVM host, forwards it to the execution engine hosting the target κ-scope , and reenters the EVMOne virtual machine to execute the relay function. There are two major steps when executing a function in EVMOne. The first step `analyze()`, functioning as a code reviewer and a bytecode parser, traverses the smart contract bytecodes, verifies the correctness, and restores a function pointer according to the bytecode function table. The second step `execute()`, regarded as a function seeker



(a) "analyze()" running time

(b) "execute()" running time



(c) contract bytecode size by "analyze()"

(d) executed bytecode size in "execute()"

Fig. 7. Execution time breakdown in EVMOne "analyze()" and "execute()" functions, and bytecode sizes in smart contract analyzed by "analyze()" and executed by "execute()".

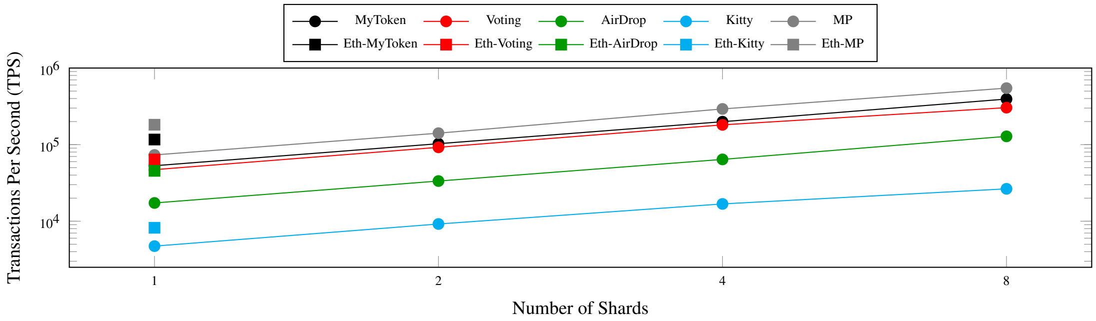


Fig. 8. Performance on 8-core Intel/Linux node

and an instruction executor, takes the analyzed code from `analyze()` and execution state as parameters, jumps to the corresponding function and execute the instructions.

Figure 7 quantifies the execution time of these two steps in our transpiler-generated Solidity codes (denoted as "ours") and their Solidity equivalents (denoted as "Solidity"), and the bytecode sizes executed in the `execute()` step for both methods. As shown in Figure 7(a) and (b), both the execution time of `analyze()` and `execute()` are increased in our transpiler-generated code. The first reason behind this is that our smart contracts have larger bytecode sizes as shown in Figure 7(c), leading to longer time of code analyzing in `analyze()`. The second reason is that our relay implementation executes more instructions in `execute`, as shown in Figure 7(d). Therefore, a more efficient way to implement the λ-relay semantics should be an extension of EVM opcodes to support the functional relay, which is left to future works.

We also carry out the performance comparison on the Intel/Linux machine. As shown in Figure 8, we can achieve 5.6x to 7.45x TPS improvements when using 8 shards over 1 shard. When comparing to the results of Solidity smart contracts, we can achieve 2x to 4.5x TPS improvements when running Crystality smart contracts on 8 shards. The experiments illustrate that our PREDA model is scalable in the testbeds of Multi-threading Transaction Processor.

6.3 On a Sharding Blockchain on Internet

To evaluate the scalability of PREDA language on distributed sharding blockchain network in real world Internet environment, we implement the sharding blockchain system, as described in Section A. The network is deployed up to 256 nodes on 128 virtual machines in different geo-locations over the world. Each virtual machine has 4 cores, 16GB memory and 50Mbps bandwidth to the public internet. Each node joins one shard. The entire network is configured to ensure that each shard has at least one participating node. In each shard, a block is created every 10 seconds in average, and each block carries transactions with restricted total consumption of computing capacity. In our experiments, computing capacity is measured by *gas burnt* as defined by EVM.

In our experiments, the computing capacity restriction, *a.k.a. gas limit*, of each block is 1024. For transactions involved in our experiments, the gas consumption of each transaction and relay transaction are following:

- In **MyToken** (Figure 4(a)): *withdraw*: 1, *deposit*: 1.
- In **Voting** (Listing 1): *vote* function: 1.
- In **AirDrop** (Listing 2): *airDrop*: 1, *deposit*: 1.
- In **CryptoKitties** (Listing 3): *breed*: 1, *lambda*#1 function: 1, *lambda*#2 function: 500, *lambda*#3 function: 1.
- In **MillionPixel** (Listing 4): *claim*: 1, *lambda*#1: 1.

Note that, we set the gas consumption in the *Lambda#2* of Crystality CryptoKitties "contract" very high because it uses the genes from the matron and sire to compute the genes of the new born, involving multiple times of square root on big numbers.

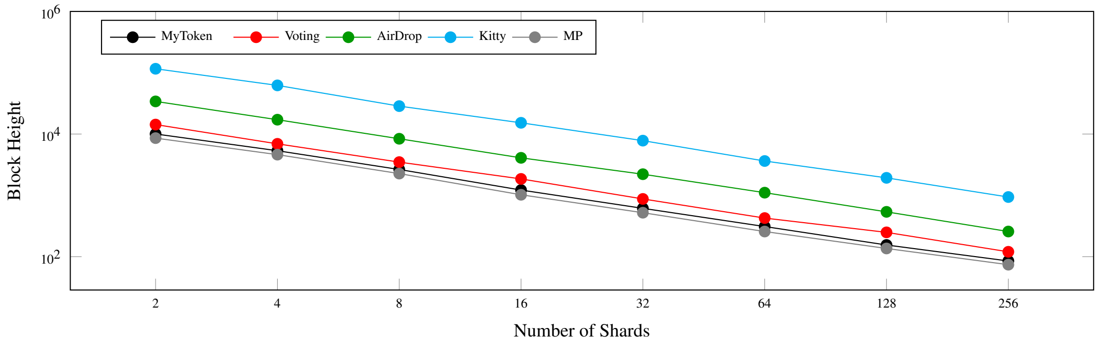


Fig. 9. Performance on distributed sharding blockchain system

Since the blockchain network operates with a fixed computing capacity, we measure the number of total block height elapsed to complete the execution of all pre-defined test transactions and all subsequentially emitted relay transactions. Figure 9 shows that smart contract developed using Crystality language scales linearly with increasing numbers of shards. Unlike simulation with multi-threading on a single node, nodes in a blockchain network share no computing resource, which avoids scalability degradation even when all nodes are fully leveraged. In average, 256 shards can yield up to 124.2x throughput improvement. Note that, based on the workload of each testing contract, we generate different numbers of testing transactions to trigger the execution of contracts. For contracts listed above, the numbers of test transactions are 10m, 10m, 400k, 100k, and 10m respectively, so that the experiment can be done within reasonable hours.

7 RELATED WORK

Many studies have optimized the performance of blockchains, e.g., those that optimize transaction dissemination and block propagation [17, 21, 23, 26, 36, 37, 39], those that accelerate consensus protocols [25, 29, 30, 34, 46, 47, 57], Layer 2 networks [40, 44, 45], etc. In this paper, we focus on sharding blockchains and parallel execution of smart contracts.

Elastico [32] is a sharding blockchain. In Elastico, a node can join a committee of a shard by solving a PoW puzzle. The committee of each shard performs PBFT to reach consensus on a set of transactions. A final committee is responsible for collecting the results of all shards, creating the final block, and sending it to the network. Elastico is designed for parallel execution of payment transactions. OmniLedger [28] is a secure, scalable, and decentralized sharding system. To ensure security, OmniLedger uses RandHound [48] as a public randomness protocol to select and assign validators to shards and to periodically rotate the assignment between validators and shards. It introduces Atomix, a two-phase client-driven “lock/unlock” protocol to ensure that a client either fully commits a transaction across shards or aborts the transaction. OmniLedger is also designed for parallel execution of payment transactions. Its pessimistic locking mechanism can lead to sequential execution of smart contract transactions when a state is shared by multiple transactions. Since neither Elastico nor OmniLedger are designed for parallel execution of smart contracts, the open-source sharding blockchain Zilliqa [53], which is “built upon the ideas of ByzCoin, Elastico, and OmniLedger”, uses a dedicated shard (i.e., a directory service committee) to process all smart contract transactions.

RapidChain [59] is a BFT-based sharding protocol that is resilient to Byzantine failures of up to a 1/4 to 1/3 fraction of the network. It does not require a trusted setup for nodes to join the network and can improve performance with a new intra-committee consensus protocol by implementing block pipelining and optimizing the gossip algorithm for large block propagation. Monoxide [55] is a PoW-based sharding blockchain that implements optimistic cross-shard transaction processing. It provides a new mining method that allows a miner to create multiple blocks for different shards with a PoW nonce, to prevent the 1% attack. OHIE [58] is a parallel chain where multiple chains execute Nakamoto consensus instances individually. To secure each chain, miners in OHIE use a Merkle tree that binds to the last blocks of all parallel chains in a newly created block. A decentralized method is also implemented to determine the total order of blocks created by parallel chains. Prism [13] improves blockchain scalability by decomposing the blockchain into multiple chains based on functionalities. It groups blockchain nodes into different chains for block proposal, voting, final block creation, etc. In [54], Prism is extended to support smart contract execution, providing the virtual machine execution module and decoupling transaction validation and state update. Miners in Prism only need to verify transactions, but not execute transactions or update states in the virtual machine. However, without sharding on-chain states, transactions accessing the states of the same contract will still execute sequentially in Prism. Chainspace [9] and COSPLIT [38] are two independent sharding blockchains that attempt to enable parallel execution of smart contract transactions. In Chainspace, the optimistic transaction execution method can lead to high abort-and-rollback overhead when there are large conflicts, which is very common between transactions invoking the same contract. In COSPLIT, on-chain states are not sharded. In addition, COSPLIT relies on the compiler to detect parallel opportunities, and many parallel opportunities are missed, such as a single transaction calling a function with a parallel loop (e.g., the `airDrop` function).

There are several studies that address parallel execution of smart contracts, but only with local parallelism, i.e., without global workload partitioning and parallel processing between nodes. Based on the transactional boosting approach [27], Dickerson et al. [20] instrument the data structures of smart contracts and detect synchronization conflicts. They allow miners to execute conflict-free transactions and update states in parallel. The execution plan information is inserted into the block by the miner. Full nodes can deterministically re-execute the received blocks. Anjana et al. [12] also use the optimistic Software Transactional Memory (STM) method to execute smart contract transactions on miners and verify blocks on full nodes, both in parallel. Based on historical data of Ethereum, Saraph and Herlihy [41] estimate the potential benefit of speculative execution and use the speculative method to execute smart contract transactions in parallel. Chen et al. [16] propose Forerunner, a constraint-based approach for speculative execution of smart contract transactions, in a pre-execution manner. All of these approaches focus on parallelizing smart contract execution on a single node. Without sharding, all nodes must continue to execute all transactions and store all on-chain states.

8 CONCLUSIONS

In this paper a novel programming model, **PREDA**, is proposed to scale out transaction processing and state hosting of smart contracts. By partitioning contract states and dividing transaction traffic, the workload of a single smart contract can be distributed to an array of execution engines working in parallel. **Programmable Contract Scope** are introduced to define the fine-grain partition boundary of contract states so that states of a smart contract can be partitioned by the underlying system with flexibility. Accordingly, **Asynchronous Functional Relay** is proposed to solving the code-data dependency by moving execution of the transaction workflow across execution engine to where involved contract states reside.

The proposed PREDA model is realized as a smart contract language, Crystality , by extending widely adopted Solidity language with new syntaxes for defining states/function in programmable contract scopes and initiating invocations of asynchronous functional relay. We demonstrate the generality and flexibility of Crystality by rewriting existing Solidity smart contract in a scalable way. In our experiments, the Crystality version of testing smart contracts are well parallelized and achieve promising linear scalability , which delivers 124.2x throughput improvement with 256 shards.

REFERENCES

[1] Another tool for language recognition. https://www.antlr.org/.
[2] Solidity airdrop smart contract, 2018. https://github.com/SpringRole/smart-contracts/blob/master/contracts/AirDrop.sol.
[3] Ehtereum client-vm connector api, 2022. https://github.com/ethereum/evmc.
[4] Etherscan api documentations, 2023. https://docs.etherscan.io/.
[5] Installing the solidity compiler, 2023. https://github.com/ethereum/solidity/blob/develop/docs/installing-solidity.rst.
[6] Million pixel dfi, 2023. https://millionpixeldfi.github.io.
[7] Solidity by example: Voting, 2023. https://docs.soliditylang.org/en/v0.8.21/solidity-by-example.html.
[8] Wrapped ether (weth), 2023. https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2.

[9] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *Proceedings of the 2018 Netowrk and Distributed System Security Symposium*, NDSS '18, 2018.

[10] Alchemy. Ethereum statistics overview 2022 – what are the top ethereum projects by total transactions?, 2022. https://www.alchemy.com/overviews/ethereum-statistics.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *Proceedings of the 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '19, pages 83–92. IEEE, 2019.

[13] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602. Association for Computing Machinery, 2019.

[14] Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. Move: A language with programmable resources, 2020. https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-\programmable-resources/2020-05-26.pdf.

[15] Vitalik Buterin and et al. A next-generation smart contract and decentralized application platform, 2013. https://github.com/ethereum/wiki/wiki/White-Paper.

[16] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 570–587. Association for Computing Machinery, 2021.

[17] Luke Dashjr. getblocktemplate updates for segregated witness. *URL: https://github.com/bitcoin/bips/blob/master/bip-0145.mediawiki*, 2016.

[18] Cameron Desrochers. A fast general purpose lock-free queue for c++, 2014. https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++.htm.

[19] Cadence Developers. Introduction to cadence, 2020. https://developers.flow.com/cadence.

[20] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 303–312. Association for Computing Machinery, 2017.

[21] Pieter Wuille Eric Lombrozo, Johnson Lau. Segregated witness (consensus layer). *URL: https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki*, 2015.

[22] The Ipsilon (ex Ewasm) team. evmone: Fast ethereum virtual machine implementation, 2022. https://github.com/ethereum/evmone.

[23] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 45–59. USENIX Association, 2016.

[24] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2315–2326, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68. ACM, 2017.

[26] Yilin Han, Chenxing Li, Peilun Li, Ming Wu, Dong Zhou, and Fan Long. Shrec: Bandwidth-efficient transaction relay in high-throughput blockchain systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 238–252. Association for Computing Machinery, 2020.

[27] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, page 207–216. Association for Computing Machinery, 2008.

[28] Eleftherios Kokoris Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Security and Privacy (SP), 2018 IEEE Symposium on*. Ieee, 2018.

[29] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 279–296, Berkeley, CA, USA, 2016. USENIX Association.

[30] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 515–528. USENIX Association, 2020.

[31] Mingzhe Li, You Lin, Jin Zhang, and Wei Wang. Jenga: Orchestrating smart contracts in sharding-based blockchain for efficient processing. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 133–143. IEEE, 2022.

[32] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[33] Arpit Mathur. Cryptokitties contract from the eth blockchain, 2017. https://gist.github.com/arpit/071e54b95a81d13cb29681407680794f.

[34] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[35] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. http://bitcoin.org/bitcoin.pdf.

[36] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. Erlay: Efficient transaction relay for bitcoin. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 817–831. Association for Computing Machinery, 2019.

[37] A. Pinar Ozisik, Gavin Andresen, Brian N. Levine, Darren Tapp, George Bissias, and Sunny Katkuri. Graphene: Efficient interactive set reconciliation applied to blockchain propagation. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 303–317. Association for Computing Machinery, 2019.

[38] George Pîrlea, Amrit Kumar, and Ilya Sergey. Practical smart contract sharding with ownership and commutativity analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '21, page 1327–1341. ACM, 2021.

[39] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. RainBlock: Faster transaction processing in public blockchains. In *Proceedings of the 2021 USENIX Annual Technical Conference*, USENIX ATC '21, pages 333–347. USENIX Association, 2021.

[40] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016. https://lightning.network/lightning-network-paper.pdf.

[41] Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts. In *Proceedings of the International Conference on Blockchain Economics, Security and Protocols*, Tokenomics '19, 2019.

[42] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *International Conference on Financial Cryptography and Data Security*, pages 478–493. Springer, 2017.

[43] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. Safer smart contract programming with scilla. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[44] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. Routing cryptocurrency with the spider network. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 29–35, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. High throughput cryptocurrency routing in payment channel networks. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI '20, pages 777–796. USENIX Association, 2020.

[46] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: Serialization of proof-of-work events: Confirming transactions via recursive elections, 2016. https://eprint.iacr.org/2016/1159.pdf.

[47] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[48] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 444–460. Ieee, 2017.

[49] Ethereum Dev Team. Solidity documentation, 2021. https://docs.soliditylang.org/en/latest/.

[50] Ethereum Dev Team. Ethereum virtual machine (evm), 2022. https://ethereum.org/en/developers/docs/evm/.

[51] Harmony Team. The harmony technical whitepaper (version 2.0), 2021. https://harmony.one/whitepaper.pdf.

[52] The NEAR Team. The near white paper, 2022. https://near.org/papers/the-official-near-white-paper/.

[53] The Zilliqa Team. The zilliqa technical whitepaper, 2017. https://docs.zilliqa.com/whitepaper.pdf.

[54] Gerui Wang, Shuo Wang, Vivek Bagaria, David Tse, and Pramod Viswanath. Prism removes consensus bottleneck for smart contracts. In *Proceedings of the 2020 Crypto Valley Conference on Blockchain Technology*, CVCBT '20, pages 68–77. IEEE, 2020.

[55] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchain with asynchronous consensus zones. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19, pages 95–112. USENIX Association, 2019.

[56] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework (draft 1), 2016. https://assets.polkadot.network/Polkadot-whitepaper.pdf.

[57] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356. ACM, 2019.

[58] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. Ohie: Blockchain scaling made simple. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, SP '20, pages 90–105. IEEE, 2020.

[59] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 931–948, New York, NY, USA, 2018. ACM.

Appendix

A SHARDING BLOCKCHAIN WITH GLOBAL SCOPE

A.1 Multi-Chain Structure

First, we use a typical single-chain structure including contract state storage, execution engine, mempool and the broadcast network, to process transactions and maintain states in global scope only, as the global chain $\mathbb{S}_{\Omega}$. Then, the system is extended by allocating additional 2^k shard chains $\mathbb{S}_{\theta i}$ with same structure as the global chain but handles transactions hosted by their own partition θi only. Parameter k is the *sharding order* controls the overall size of the sharding system, exponentially, which makes total number of shards to be a power of two.

In the system, block creations are synchronous. For each new block generated in global chain, one, and only one, block per shard chain will be generated, which results in aligned block heights for all chains in the system. In any node participated in one or more shards, the block of global chain at height h shall be received and executed after any block at height $h-1$ is executed, and prior to any block at height h in any shard chain, which provides a consistent view of the global scope ϕ_{Ω} at height h throughout the entire network when executing any transaction in shard chains. In each node, the contract states in global scope is available for thread-safe read-only access by transactions executing in shard chains.

A.2 Data Structures

A shard chain doesn't have own consensus proof, instead it inherits consensus proof from the global chain. Figure 11 illustrate key data structures that extend a single-chain blockchain system with shard chains. Existing data structures, block header and block body, of the single-chain system are denoted here as *consensus header* and *global block*. The consensus header carries the proof for a validated consensus proof (e.g. the PoW nonce, or the aggregation of PoS signatures) and the hash pointing to the global blocks θ_g, which carries actual transactions in global scope being confirmed. The two data will be broadcasted in the global broadcast network that all nodes in the network will receive those regardless of the sharding division. Every node thus has the contract states of the global scope and keeps updating.

To extending the global chain with shard chains, two additional aggregated proofs (Θ and Υ) are introduced and embedded in the consensus header at every block height to prove validities of all newly generated shard blocks and emitted relay transactions at that height h.

- A Merkle tree Θ is built by taking hashes of 2^k shard blocks at height h of all shard chains. The Merkle tree root will be embedded in the consensus header so that a shard block θ_s can be verified in any shard chain.

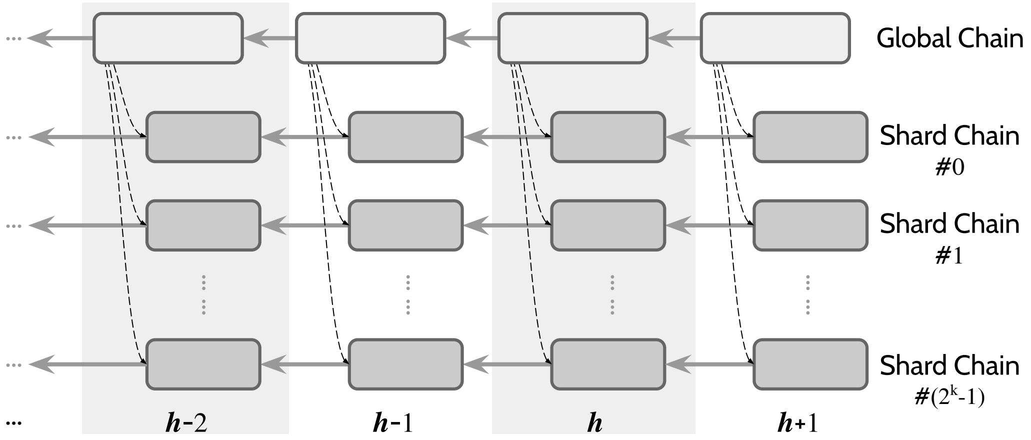


Fig. 10. A sharding blockchain system with a global chain and multiple shard chains.

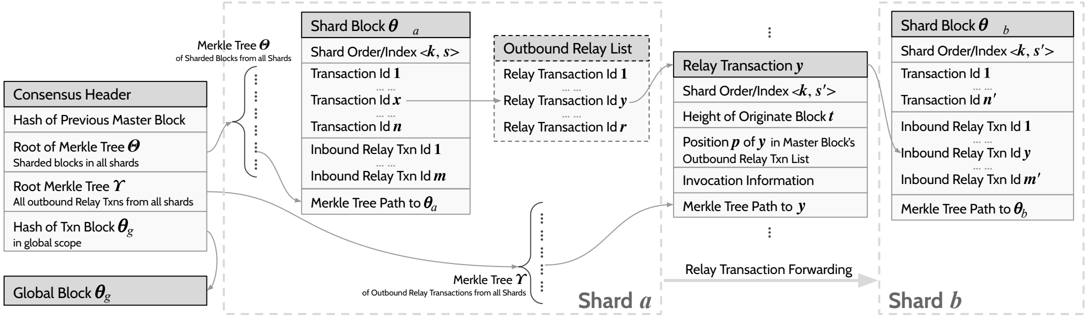


Fig. 11. In our synchronous sharding, a block contains a consensus header, a global transaction block, and transaction blocks, one for each shard. Relay transactions are verified with the merkle tree root carried by the consensus header.

- A Merkle tree Υ is built by taking hashes of relay transactions emitted by blocks at height h of all shard chains to facilitate functional relays. Embedding the root of the Merkle tree Υ in every consensus header enables validation of any inbound relay transactions received in the global chain or in any shard chains, by checking upon the Merkle root carried by the consensus header at the emitted block height of a particular relay transaction.

B CRYSTALITY SMART CONTRACTS

Voting is a smart contract that allows voters to vote on the candidate proposals [7]. In Solidity code, the candidate proposals are defined in a global array and voting transactions from voters are executed one by one in the EVM to read and modify the proposals accordingly. The equivalent Crystality Voting smart contract is shown in Listing 1. We define the variable `proposals` in the global scope ϕ_Ω with the keyword `@global` and introduce a new variable `votes` in the engine scope with the keyword `@engine`. The basic idea is to use this engine scope variable as an intermediate layer to obtain voting results in each engine, and aggregate all partial results to the final results defined in the global scope.

```
contract Ballot {
  Proposal[] @global proposals;
  uint64[] @engine votes;
  bool @address voted;

  function vote(uint32 proposal) @address public returns (bool) {
    if (proposal < proposals.length) {
      votes[proposal]++;
      voted = true;
      return true;
    }
    return false;
  }

  function finalize() @address public {
    require(controller == msg.sender);
    relay @engines {
      relay @global (votes) {
        for (uint32 i = 0; i < votes.length; i++)
          proposals[i].weights += uint64(votes[i]);
      }
    }
  }
```

```
}
```

Listing 1. The Voting smart contract in Crystality

When receiving a transaction from an address, the function `vote()` defined in the κ-scope (i.e., the address scope using `@address` in this case) directly updates the engine scope variable `votes`. Concurrent voter transactions partitioned by address scope will be forwarded to different engines and modify different instances of the variable `votes`. When the function `finalize()` is called to complete the voting process, the function sends a relay to all engines and triggers all engines to send a relay to the global with the partial voting results, i.e. `votes`. The global scope variable `proposals` will be updated and finalized.

AirDrop is a smart contract sending tokens or NFTs from an address to a group of destination addresses [2]. In Solidity code, the transfers occur in a `for` loop, and the global states of `balances` are updated sequentially. This could be the performance bottleneck and exacerbated if computations in the loop involve heavier operations such as divisions of big numbers and hash functions. Existing work using multithread execution [16, 41, 42] or sharding execution [9, 38] cannot parallelize such a case because none of them can leverage parallel opportunities inside a single transaction.

```

contract AirDrop {
  uint @address balance;

  function airDrop(Payment[] memory recipients) @address public returns (bool) {
    uint total = 0;

    for (uint i = 0; i < recipients.length; i++) {
      require(recipients[i].amount >= 0);
      total += recipients[i].amount;
    }

    if (total > balance) return false;
    balance -= total;

    for (uint i = 0; i<recipients.length; i++) {
      if(recipients[i].amount > 0) {
        relay @recipients[i].to (amount) {
          balance += amount;
        }
      }
    }
  }
}

```

Listing 2. The AirDrop smart contract in Crystality

The equivalent Crystality AirDrop smart contract is shown in Listing 2. In the `for` loop of Crystality `AirDrop()` function, a relay is issued for each recipient. Multiple relays will be executed in different target κ-scopes keyed by recipient address.

CryptoKitties is a blockchain game enabling players to breed and trade virtual cats through NFTs on Ethereum [33]. Upon its launch time in October 2017, the game accounted for a substantial 10% of Ethereum's network traffic, causing a surge in gas prices that temporarily disrupted Ethereum's usability. We implement the Crystality CryptoKitties smart contract as shown in Listing 3, focusing on the `breed()` function. The execution of this function will trigger a sequence of chained relays, from the end-user's κ-scope to the matron kitty's κ-scope, the sire kitty's κ-scope, and ultimately back to the end-user's κ-scope.

```

contract KittyBreeding {
  KittyInfo[] @global allKitties;
  KittyInfo[] @engine newBorns;
  mapping<uint32 => Kitty> @address myKitties;

  function breed(uint32 m, uint32 s, bool gender) @address public {
    require(m < allKitties.length);
    require(s < allKitties.length);
    require(allKitties[m].gender);
    require(!allKitties[s].gender);

    relay @allKitties[m].owner (m, s, gender) {
      myKitties[m].lastBreed = block.number;

      relay @allKitties[s].owner (myKitties[m].genes, m, s, gender) {
        uint new_gs = genesMix(myKitties[m].genes, myKitties[s].genes);

        relay @msg.scope (m, s, gender, new_gs) {
          uint birth_time = block.number;
          uint id_nb = newBorns.length | (1 << 255);
          _addNewKitty(msg.scope, new_gs, id_nb, m, s);
          KittyInfo memory n;
          n.gender = gender;
          n.birthTime = birth_time;
          n.owner = msg.scope;
          newBorns.push(n);
        }
      }
    }
  }
}

```

Listing 3. The CryptoKitties smart contract in Crystality

Similar to the Crystality Voting smart contract, an engine scope variable `newBorns()` is defined. Using this variable, write operations on chain are executed on different engines independently and simultaneously. Although the smart contract accesses the global scope variable `allKitties`, this doesn't become a global barrier because the contract function only reads the global scope variable.

MillionPixel is a DeFi smart contract enabling users to buy pieces of DeFi Meta Chain's history [6]. We implement this smart contract in Crystality to show that how to define and use a κ-scope other than the address scope. As shown in Listing 4, we use the `uint32` κ-scope as the relay target, denote as `@unit32`.

```

contract MillionPixel {
  Land @uint32 land;

  function claim(uint16 x, uint16 y) @address public {
    uint32 index = uint32(x) * 65536 + uint32(y);
    relay @index (msg.sender) {
      if (true == land.flag) return;
      land.flag = true;
```

```
        land.owner = msg.sender;
      }
    }
  }

```

Listing 4. The MillionPixel smart contract in Crystality