Crystality: A Programming Model for Smart Contracts on Parallel EVMs

Hao Wang[†], Minghao Pan[†], Jiaping Wang[†§*]
[†] International Digital Economy Academy (IDEA), China
[§] The Hong Kong University of Science and Technology (Guangzhou), China
{wanghao2020,panminghao,jiapw}@idea.edu.cn

Abstract

Scaling blockchain performance through parallel smart contract execution has gained significant attention, as traditional methods remain constrained by the performance of a single virtual machine (VM), even in multi-chain or Layer-2 systems. Parallel VMs offer a compelling solution by enabling concurrent transaction execution within a single smart contract, using multiple CPU cores. However, Ethereum's sequential, shared-everything model limits the efficiency of existing parallel mechanisms, resulting in frequent rollbacks with optimistic methods and high overhead with pessimistic methods due to state dependency analysis and locking.

This paper introduces Crystality, a programming model for smart contracts on parallel Ethereum Virtual Machines (EVMs) that enables developers to express and leverage the parallelism inherent in smart contracts. Crystality introduces **Programmable Contract Scopes** to partition contract states into non-overlapping, parallelizable segments and decompose a smart contract function into finer-grained components. Crystality also features **Asynchronous Functional Relay** to manage execution flow across EVMs. These features simplify parallelism expression and enable asynchronous execution for commutative contract operations.

Crystality extends Solidity with directives, transpiling Crystality code into standard Solidity code for EVM compatibility. The system supports two execution modes: an asynchronous mode for transactions involving commutative operations and an optimistic-based fallback to ensure block-defined transaction order. Our experiments demonstrated Crystality's superior performance compared to Ethereum, Aptos, and Sui on a 64-core machine.

CCS Concepts: • **Computing methodologies → Parallel programming languages**.

*corresponding author

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710879

Keywords: Blockchain, Smart Contract, Parallel Execution, Concurrency Control, Parallel EVMs

1 Introduction

Since the advent of Bitcoin [38], enhancing blockchain performance has been a major focus in both academia and industry. Recently, parallel VM techniques [10, 14, 22, 23, 26, 45, 46] have shown great potential in increasing smart contract execution throughput, achieving rates of hundreds of thousands of transactions per second (TPS). These mechanisms enable concurrent execution of smart contract transactions across multiple VMs after consensus, without altering the underlying consensus and network layers. As a result, they are quickly adopted by recent public blockchains such as Solana [21], Aptos [20], Sui [54], Sei [32], and Monad [41].

Blockchains using parallel VMs typically adopt either optimistic or pessimistic parallelization strategies. Optimistic systems, like Aptos, Sei (V2), and Monad, ignore state dependencies, execute and validate transactions concurrently, and re-execute them in case of conflicts. Pessimistic systems, including Solana, Sui, and Sei (V1), pre-check transactions before execution, analyze state dependencies, and execute only non-conflicting transactions concurrently. Both approaches have their well-known limitations: optimistic systems may encounter frequent re-executions due to heavy contentions on shared states, while pessimistic systems incur significant overhead in state analysis and locking.

Moreover, existing parallel mechanisms do not fully exploit the potential of parallel VMs. Consider two scenarios. Firstly, consider two transactions $\{tx_i : a\text{->}b\}$ and $\{tx_{i+1} : c\text{->}b\}$, both accessing state b at the same address. Current mechanisms require tx_i to complete and commit its update to state b before tx_{i+1} can run. However, if these transactions invoke a token transfer function, the execution order of deposits (i.e., additions) on state b does not matter. An asynchronous execution method would be more efficient in such cases where order is not important.

Secondly, if tx_i and tx_{i+1} invoke a voting contract where state b represents the final result, optimistic systems might either abort and re-execute transactions sequentially (like pessimistic systems), or do not abort but update state b sequentially, if aware of the commutative property of addition. A more efficient strategy would allocate a temporary state b_i in each VM_i, execute transactions concurrently on states b_i,

and then aggregate all b_i into the final state b. Existing mechanisms cannot adequately express the inherent parallelism in such a case, because popular smart contract languages, such as Solidity [52] and Move [13], using sequential and shared-everything models, are not suitable for parallelism.

In this paper, we introduce **Crystality**, a programming model for smart contracts on parallel EVMs. The core idea is to partition contract states into non-overlapping, parallelizable segments and decompose contract functions into finer-grained components based on state dependencies. By targeting individual states instead of entire transactions, programmers can easily express parallelism, while allowing the underlying system to enforce asynchronous execution. We present **Programmable Contract Scopes** as directives to define state partitioning and function decomposition, and **Asynchronous Functional Relay** to manage transaction logic across VMs. Crystality's parallel EVM system supports two execution modes: an asynchronous, fast mode for commutative operations and an optimistic fallback mode that preserves transaction order. Using consensus-layer information for correctness verification, Crystality seamlessly switches between these modes without additional overhead. This combines the fine-grained parallelism of the PGAS model [16] with the transactional guarantees of the STM model [47].

Crystality's directives are implemented in Solidity, with a two-stage compilation process that generates standard Solidity code and EVM bytecode. We implement five smart contracts using Crystality: TokenTransfer, Voting, AirDrop, CryptoKitties, and MillionPixel. On a 64-core machine, Crystality achieved speedups of 17x-42.9x over single-threaded Solidity. Compared to Aptos and Sui, Crystality demonstrated superior scalability, achieving 34.59x-53.36x speedups on 64 EVMs, while Aptos and Sui achieved 3.29x-19.46x and 12.1x-20.22x speedups over their own baselines, respectively.

Our main contributions in this paper include:

- Introducing Crystality: A parallel model for smart contracts on parallel EVMs, the first of its kind.
- Developing Crystality's System: A parallel EVM system with two execution modes–one for transactions involving commutative operations and another for more general cases–along with a transpiler that generates standard EVM bytecode, ensuring full compatibility.
- Comprehensive Evaluation: A detailed comparison of Crystality with Solidity/Ethereum (sequential execution), Aptos (optimistic parallelism), and Sui (pessimistic parallelism) across various contracts and workloads on a 64-core machine, demonstrating significant improvements in throughput and scalability.

2 Background

2.1 Blockchain and Smart Contract

Figure 1 presents the high-level architecture of a blockchain node, comprising three layers. The network layer handles

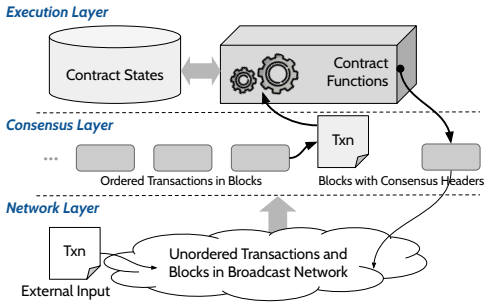

Figure 1. The unified architecture of a blockchain node.

the transmission of transactions and blocks via a broadcast network. Transactions are cached in a transaction pool (memory pool) until they are inserted into an ordered transaction block for execution. The consensus layer orders transactions, typically by gas fees, and forms them into blocks. The execution layer contains the execution engine, processing transactions from the ordered block and updating contract states as needed. Once finalized by the consensus layer, the block is propagated across the network. Non-creator nodes receive blocks, execute transactions, and update contract states after verifying consensus.

A smart contract consists of states (variables) and behaviors (functions). Figure 2(a) showcases a Solidity ERC20 contract, with a `balances` variable for user balances and a `transfer` function to transfer a specified `amount` of tokens from the sender (`msg.sender`) to a `payee`. In Ethereum, each node stores compiled contract bytecode and executes transactions (i.e., invoking contract functions) in its EVM [53]. The EVM host manages states and blocks, interfaces with the EVM and key-value store, and communicates with the underlying layers.

Notably, existing parallel VM systems [20, 21, 32, 41, 54] focus on parallelizing transaction execution within a single node's VMs, avoiding cross-node parallelization. This approach aligns with the blockchain community's preference for simpler consensus protocols, as cross-node parallelization requires significant consensus modifications. This paper follows that trends, exploring intra-node parallelism.

2.2 Transaction Execution on Parallel VMs

Figures 3 (a) and (b) depict the core architectures of Aptos and Sui, two blockchain systems that leverage parallel VMs but differ in concurrency control. Aptos uses optimistic concurrency control (OCC) [31] and Sui uses pessimistic concurrency control (PCC).

In Aptos (Figure 3(a)), the main thread initializes shared data structures, including a transaction hashmap, an execution task (`E-Task`), a validation task (`V-Task`), a transaction dependency vector (`Txn-Deps.`), and a hashmap for the `read-set` and `write-set`. Worker threads execute and validate transactions on Aptos Move VMs, handling tasks such as

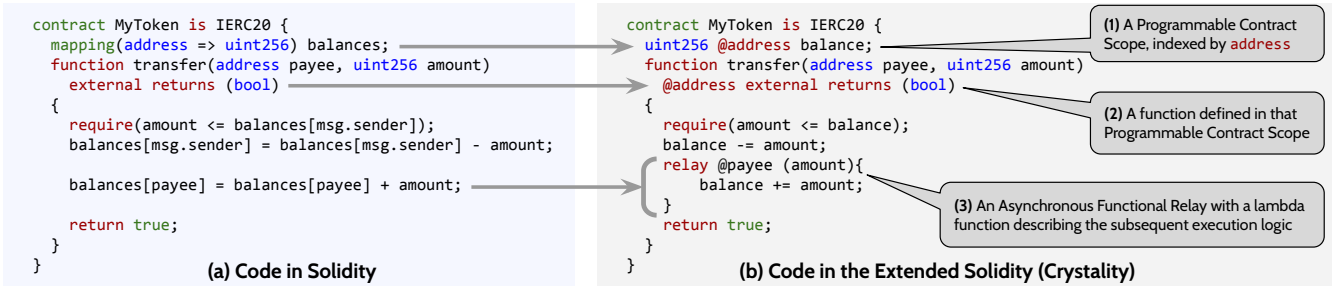

Figure 2. A glance at the ERC20 token transfer smart contract in Solidity and in Crystality, respectively.

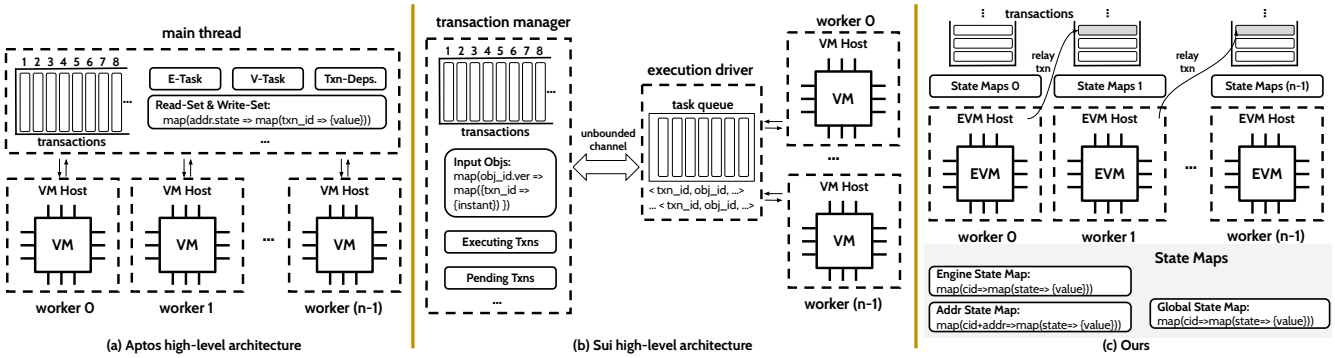

Figure 3. The architectures of Aptos, Sui, and Crystality. Aptos implements optimistic parallelization while Sui uses pessimistic parallelization for smart contract execution. Crystality executes transactions concurrently on partitioned states, with the guidance from programmers.

signature verification and contract function execution. Modified states are recorded in the `write-set`, and Multiversion Concurrency Control (MVCC) allows multiple state versions. After execution, transactions move to `V-Task` for validation, where workers check for conflicts in the `read-set`. Conflicts trigger transaction abortion and re-execution.

In Sui (Figure 3(b)), PCC is centralized. Key structures include the `InputObjs` hashmap for accessed states, `Executing-Txns` hashmap for ongoing transactions, and `PendingTxns` hashmap for those awaiting dependencies. States are tracked by object ID and version number. Only non-conflicting transactions proceed to `ExecutionDriver` for parallel execution.

2.3 Problem Statements

Both OCC and PCC approaches face notable limitations in smart contract execution. OCC is effective for read-heavy workloads typical of databases and Big Data systems, where writes are relatively rare. However, blockchain transactions are predominantly write-heavy, as read-only queries are handled off-chain by centralized services like Etherscan [8] and BTCScan [7]. Consequently, OCC suffers from frequent rollbacks and suspensions, leading to significant overhead. PCC, while more robust for compute-intensive tasks, rigorously checks state dependencies and locks conflicting transactions. This approach is ill-suited for smart contracts with simple, lightweight operations, such as ERC20 token transfers, where the added overhead can become a bottleneck.

Current parallel VM systems rigidly enforce block-defined transaction order, failing to leverage the commutative property of many contract operations, such as token transfers, votes, airdrops, and others. In fields like operating systems [15], databases [30], and Big Data systems [61], the commutative property is exploited for concurrency. However, existing parallel VM systems struggle to apply it effectively due to their coarse-grained transaction-level execution, often involving multiple states. Moreover, existing programming models, such as Solidity and Move, lack mechanisms to easily express fine-grained parallelism within a smart contract or manage asynchronous execution flow across VMs.

3 System Design

Our solution introduces a parallel programming model to express contract-level parallelism and optimize asynchronous and pipelined execution in parallel VMs. Key design considerations include: First, partitioning contract states non-overlappingly across VMs, typically by address. Second, directing transactions to the VM holding the relevant state and chaining execution across VMs for functions accessing multiple states. Third, maintaining globally shared states accessible to all VMs with concurrency control. Fourth, supporting parallel algorithms for states shared by specific user groups. From these principles, two core functionalities emerge: defining state scopes to guide partitioning, and enabling controlled execution switching between VMs.

In Solidity, all contract state variables and functions are defined globally, allowing unrestricted access to all states, regardless of dependencies. This shared-everything model hampers efficient concurrency control in parallel VM systems. Moreover, Solidity lacks support for workflow switches from processing one state to another. Thus, our first step is to extend Solidity by introducing Crystality's directives.

We introduce **Programmable Contract Scope (κ-scope)**, enhancing Solidity's ability to describe contract state partitioning. A programmable scope, denoted as ϕ, comprises a set of variables $\{\mathcal{S}\}$ and functions $\{\mathcal{F}\}$ with restricted access to variables within the same scope. These scopes are indexed by a key k with a built-in type, typically an address type. The set of keyed κ-scopes Φ is formulated as:

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

Here, $\mathcal{K}$ denotes the set of all possible values of k.

A function $f_{\phi_k} \in \mathcal{F}$ of a κ-scope ϕ_k has immediate access to variables $\mathcal{S}$ and functions $\mathcal{F}$ within that κ-scope, besides invocation arguments and execution context (e.g., block height and message sender). Unlike Solidity, a function in Crystality is invoked by specifying the current κ-scope (**target κ-scope**) to start.

We propose **Asynchronous Functional Relay (λ-relay)** to decompose transaction execution into multiple invocations of these scope-narrowed functions across VMs, ordered by data dependency. To continue the execution dealing with state in another κ-scope $\phi_{k'}$, an asynchronous invocation of a function $g_{\phi_{k'}}$ is initiated, formulated as a λ-relay:

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

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

There are three typical scopes. **Address Scopes**: when k in κ-scope ϕ_k is of the address type, the κ-scopes are defined as address scopes, dividing contract states among VMs by address, with each VM handling a subset of addresses. **Engine Scopes**: An instance of κ-scope $\phi_{\theta(i)}$ is built-in for each execution engine instance, representing a scope for immediate read/write by any function executed in the current VM. Engine scopes define local-shared states. **Global Scope**: A built-in κ-scope ϕ_Ω is logically singleton across all engines. Its states are consistently updated by all VMs, providing a scope available for immediate read access by any function.

In Crystality, directives **`@address`**, **`@engine`**, and **`@global`** define these scopes. Programmers use these directives to reimplement smart contracts. Figure 2(b) illustrates the Crystality version of the ERC20 contract. In part (1), κ-scopes keyed by `@address` type define users' balances, providing a fine-grained separable states description. All κ-scopes share the same variable definition but each has a unique instance. The transfer function is defined in the same set of κ-scopes in part (2), invoked by providing k (i.e., the payer's address) as the target κ-scope. In part (3), to proceed with the deposit to the payee k' after a withdrawal, a λ-relay is initiated with $\phi_{k'}$ as the target κ-scope, adding funds to the payee's balance and executed by an engine that hosts the state of $\phi_{k'}$.

3.1 Partitioning

In Crystality, programmable scopes define fine-grained boundaries for partitioning contract states. The actual partitioning strategy requires coordination with the execution engine. Key considerations include:

- A partition scheme should evenly partition the entire value space of k without overlap, uniquely mapping each k to an execution engine instance.
- Partition mapping should be resolved solely based on the k of a κ-scope ϕ_k, ensuring a single execution engine instance without ambiguity.
- Contract states in storage are indexed by k and are written according to the current κ-scope ϕ_k.
- Upon initiating a λ-relay, the host should convert the relay into a local invocation, if its target κ-scope is hosted by the current execution engine.

On a blockchain with 2^m VMs, partitioning by address can use the first m-bits of an address a. A user-initiated transaction is directed to the VM by the Address-to-VM mapping, as $(a \& (2^m - 1))$.

3.2 Relaying

A function executing in a κ-scope ϕ_k must initiate a λ-relay to continue execution for accessing the state in another κ-scope $\phi_{k'}$. A λ-relay can be converted to a relay transaction by the execution engine and sent to the destination. The actual mechanism for passing a λ-relay requires coordination with the engine, which should have following capabilities:

- A λ-relay can be converted to a relay transaction with target κ-scope, the identifier of the function to be invoked, and any associated arguments.
- A relay transaction can be transferred to the destination engine, where it will be ordered deterministically with other relay transactions before execution.
- Unlike cross-chain or cross-shard relays (e.g., Polkadot [58] and NEAR [55]), Crystality's relay transactions are not involved in the consensus layer, so a proof (e.g., Merkle path) for relay verification is unnecessary.

A λ-relay to the global scope ϕ_Ω follows the same logic as normal λ-relays. In implementation, we use a shared map by all VMs for optimization (`GlobalStateMap` in Figure 3(c)).

3.3 Parallel Execution Framework

Figure 4 compares the transaction-level parallelism used in Block-STM [23] and other OCC-based solutions and Crystality's state-level parallelism. Consider n token transfer transactions, each involving a sender state S_i and a common receiver state S_x ($\{Tx_i : S_i \text{->} S_x\}$), executed on p (e.g., 4) parallel VMs P_0-P_3. In optimistic parallelism, at time t_0, P_0 executes

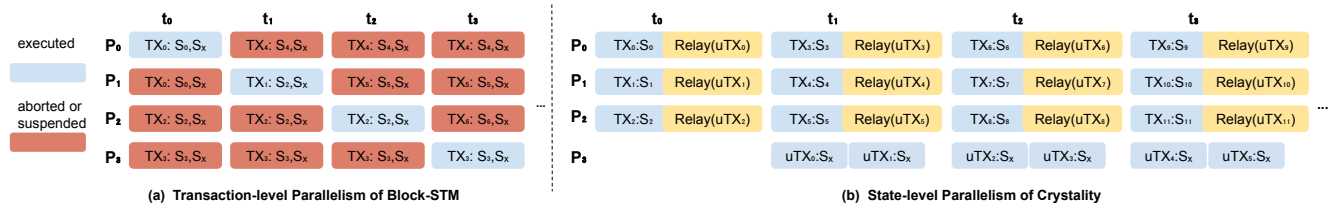

Figure 4. Comparison of Block-STM's transaction-level parallelism and Crystality's state-level parallelism.

Tx_0, P_1 executes Tx_1, etc. Due to conflicts at S_x, only P_0's transaction commits, while Tx_1-Tx_3 are aborted. At time t_1, P_0 processes Tx_4, while P_1-P_3 re-execute Tx_1-Tx_3. OCC approaches might abort Tx_2-Tx_4, since Tx_1 has the smallest ID; Block-STM suspends Tx_2-Tx_4 as Tx_1 marks S_x as estimated during validation at t_0. For n such transactions, Block-STM faces at least $n * (p-1)/p$ abortions or suspensions.

In Crystality, parallelism is managed at the state level. Assume P_3 exclusively updates S_x, and P_0-P_2 handle other states. The time taken to generate and place a relay in the destination VM is similar to or less than processing a state (as confirmed in our experiments). At t_0, P_0 processes S_0 and generates $Relay(uTx_0)$ for S_x, P_1 processes S_1 and generates $Relay(uTx_1)$, etc. At t_1, P_0-P_2 process S_3-S_5 and generate $Relay(uTx_3)$-$Relay(uTx_5)$, while P_3 sequentially processes $Relay(uTx_0)$ and $Relay(uTx_1)$ to update S_x. This forms a pipeline between processing sender states, generating/placing relays in destination VMs, and processing receiver states. This mechanism has three major benefits: (1) No locks are required during state updates, as each VM only accesses its partitioned states. (2) Single state processing: each state is processed once, eliminating abortions, suspensions, and re-executions. (3) Minimal locking overhead: locking occurs only when placing relays in destination VMs, but the overhead is effectively hidden within the pipelined execution.

Figure 3(c) presents our high-level architecture. Each EVM maintains its own `AddrStateMap` and `EngineStateMap` for partitioned on-chain states and uses a `ConcurrentQueue` for transactions. States in the maps are read and written by their EVMs, while a cross-EVM state access is handled via a relay.

3.4 Two Execution Modes

Algorithm 1 describes the parallel and asynchronous execution in Crystality's parallel EVM system. The first two functions map contract states and original transactions in a block to EVMs using an Address-to-VM mapping derived from the user's address. The third function outlines asynchronous execution. Lines 22, 26, 28, using `Sort()` and `Barrier()`, are unnecessary for asynchronous execution, but their purpose will be explained later. In each EVM, while the receive queue rq_x is not empty, the worker thread executes transactions in rq_x sequentially. Since a contract function is decomposed into multiple components, a relay $R_{i,j}$ may be issued when accessing a state in another EVM. The relay is buffered in the send queue sq_x. After Tx_i's execution is completed, its relays are dispatched to the corresponding EVM_y. This process continues until all receive queues are empty, and the main thread retrieves the next block from the consensus layer.

Algorithm 1 Parallel and Asynchronous Execution

1: **Input:**
2: m: Number of EVMs
3: rq_x: Receive concurrent queue of EVM_x
4: sq_x: Send queue of EVM_x
5: $CState = \{CState_1, \ldots, CState_m\}$: Partitioned contract states
6: $f(a)$: Address-to-EVM mapping func. $f(a) = (a \,\&\, (2^m - 1))$
7: n: Block size
8: k: Maximum relays a transaction can issue in one iteration
9: t: Maximum depth in a nested relay
10: Tx_i: Original txs in a block, $i \in \{0, 1, \ldots, n-1\}$
11: $R_{i,j}$: A relay tx issued by Tx_i, where $j < k$
12: **procedure** Contract State Partitioning
13: **for** each $CState_x$ in $CState$ **do**
14: Assign $CState_x = \{CState_a \mid f(a) = x\}$ to EVM_x
15: **procedure** Original Transaction Routing
16: **for** each Tx_i **do**
17: Determine EVM_x using sender's addr. a: $EVM_x = f(a)$
18: Route Tx_i to EVM_x
19: Tx_i is cached in EVM_x's receive queue rq_x
20: **procedure** Processing Transactions in EVM_x
21: **while** rq_x is not empty **do**
22: **Sort** Txs in ascending order of tx index value (Eq.3)
23: **for** each Tx_i **do**
24: Execute Tx_i
25: For any relay $R_{i,j}$ accessing $CState_b$, cache it in sq_x
26: **Barrier()**
27: Dispatch $R_{i,j}$ to $EVM_y = f(b)$
28: **Barrier()**

This algorithm operates in a purely asynchronous mode, making it fast and efficient for transactions involving commutative operations. However, if relays are also ordered, the results will match those of sequential execution, even for non-commutative operations in many cases. Such cases are common in GameFi and SocialFi smart contracts, such as MillionPixel, where user-initiated transactions compete to tag a coordinate with ownership and results depend on execution order. To reduce the need to switch to Algorithm 2, `Sort()` and `Barrier()` are used. In Lines 26 and 28, all threads synchronize using a barrier (by `event()` and `wait()`), awaiting the completion of relay dispatch. At the beginning of each

iteration, transactions in rq_x are sorted in ascending order by transaction index value, calculated using Equation 3.

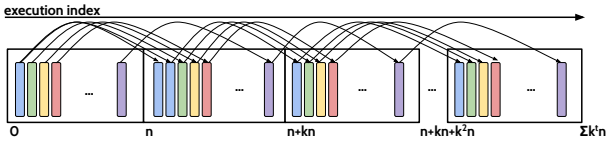

Figure 5. Deterministic execution order of relays.

Figure 5 shows a deterministic execution order of relays iteratively. Let n represent the block size (the number of ordered original transactions in a block), k the maximum number of relays a transaction can issue in one iteration, and t the maximum nesting depth. Initially, the original transactions' execution indices match their order in the block. In the first relay iteration, there are up to $n \cdot k$ relays; in the second iteration, up to $n \cdot k^2$ relays; and so on, until the t-th iteration with a maximum of $n \cdot k^t$ relays. In this index space, a transaction is denoted as $< i_0, i_1, ..., i_t >$, where a relay $< i_0, i_1, ..., i_j >$ is the i_j-th relay of its parent $< i_0, i_1, ..., i_{j-1} >$. The transaction index value is calculated as:

$$I = \Sigma_{j=0}^{t-1} n \cdot k^j + \Sigma_{j=0}^{t} i_j \cdot k^j \quad (3)$$

For example, with $n = 10000$, $k = 256$, and $t = 8$, transaction $< 900, 0, 1 >$, which is the second relay of transaction $< 900, 0 >$ (itself the first relay of the transaction 900), has a value of 59,239,401. This value is used to sequence relays. As real-world scenarios issue far fewer relays, we use `uint128_t` type to hold the transaction index value.

Incorporating transaction index values, Algorithm 1 offers a deterministic method for parallel execution, but does not strictly enforce block-defined transaction order. For example, consider $\{tx_i : a \rightarrow b\}$ and $\{tx_{i+1} : b\}$, both modifying state b of the same address. When tx_{i+1} updates state b directly, the relay from tx_i might execute later, leading to incorrect state updates. To address such cases and ensure correct commitment order, we introduce a stricter execution approach.

Algorithm 2 guarantees block-defined transaction order. It is a refined version of OCC that re-executes only the conflicting relay chain (a tree of relays rooted in the conflicting relay) rather than the entire transaction, enabling finer-grained control. The key idea is that during each transaction's execution, including its relays, worker threads notify the coordinator thread about Tx_i and its relays. Upon completion, the coordinator thread commits the state updates of the transaction.

Lines from 18 to 31 detail the coordinator's state commitment process. If the coordinator receives a notification for Tx_i, it adds the change of Tx_i to the commitment queue cq. The coordinator then checks if the transaction Tx_i's index matches the current commitment order and verifies if all relays associated with Tx_i are received. A transaction relay tree is built with the notifications to track the relay-chain of Tx_i, where each node (i.e., relay) is validated via the `Validate` function. If valid, the state update is finalized; if not, the subtree rooted at the invalid node is cut, and the relay is re-executed, triggering its subtree relays.

Algorithm 2 Optimistic-based Parallel Execution

```
1:  Input:
2:  I: transaction index as I = < i_0, i_1, ..., i_t >
3:  key: cid.state for engine- and global-state, (cid + addr).state
    for addr-state
4:  MVS: multi-version storage; its value is a map in descending
    order of transaction index: map(key => map{I, value})
5:  Txs: an index-to-transaction map: map(I => transaction)
6:  cq: a priority queue in coordinator thread for transaction com-
    mitment, in ascending order of transaction index
7:  co: commitment order, co ∈ {0, 1, ..., n − 1}
8:  procedure Processing Transactions in EVM_x
9:      while rq_x is not empty OR Not terminated do
10:         Sort Txs in ascending order of transaction index I
11:         for each Tx_i do
12:             Execute Tx_i
13:             For any relay R_{i,j} accessing CState_b, cache it in sq_x
14:             End Tx_i Execution
15:             Notify coordinator with args (Tx_i, its relays R_{i,j})
16:         Dispatch R_{i,j} to VM_y = f(b)
17: procedure Committing States in Coordinator
18:     Wait updates from any Tx
19:     Put Tx_i into cq
20:     if Tx_i.I.i_0 == co AND All relays of Tx_i received then
21:         tree = BuildTxnRelayTree(Tx_i, cq)
22:         CommitFlag = true
23:         for Each node in tree do
24:             if Validate(MVS,node) then
25:                 Finalize(MVS,node.WriteSet)
26:             else
27:                 CommitFlag = false
28:                 Cut subtree with node as root; Re-execute node
29:         if CommitFlag then CommitmentOrder += 1
30:     if CommitmentOrder == n then
31:         Terminate all worker threads
32: procedure GetLatestCommittedVersion(MVS,I,key)
33:     multi_version_values = MVS.find(key)
34:     for each value in multi_version_values do
35:         if value.I<I then return value
36:     return NULL
37: procedure Validate(MVS,txn)
38:     read_set = txn.ReadSet
39:     for each key in read_set do
40:         value = read_set.get(key)
41:         v = GetLatestCommittedVersion(MVS,txn.I,key)
42:         if v.value != value then return false
43:     return true
```

The `GetLatestCommittedVersion` function retrieves the most recent committed version of a given key from the multi-version storage `MVS`. It iterates through the versions and returns the highest index version less than the current transaction index `I`. Otherwise, it returns `NULL`. The comparator

for `I` is overloaded to compare i_j in $< i_0, i_1, ..., i_t >$ one by one, rather than comparing the transaction index value.

The `Validate` function ensures a transaction's states consistency with the latest committed versions in `MVS`. It retrieves the read set of the transaction, and compares each key's value in the read set with the latest committed version of that key. If any value doesn't match, the function returns `false`, indicating that the transaction is invalid. If all values are consistent with the latest committed versions, it returns `true`, confirming the transaction's validity.

In the implementation, a key challenge is detecting when Algorithm 1 produces incorrect results and triggers a switch to Algorithm 2. This is resolved by leveraging data from the consensus layer. In blockchains, nodes verify execution outcomes using Merkle roots in the block header, generated from transaction and state tries (e.g., via Merkle Patricia Trie [17]). After executing a block, full nodes must verify these roots to ensure that transactions executed and states updated match the block creator's output. Thus, **on full nodes**, we use the Merkle roots to check whether Algorithm 1 produced incorrect results. If discrepancies are detected, full nodes revert to Algorithm 2 for re-execution. **On block creators**, Algorithm 2 can be directly used, as transaction execution delays are masked by consensus overhead (e.g., mining or communication delays within consensus committees).

Ensuring atomicity is another critical aspect. Algorithm 2 guarantees atomicity through a coordinator thread that commits states. Algorithm 1, however, does not inherently offer this guarantee. In blockchains, atomicity is typically enforced by state reversion, such as Solidity's `revert()` function. To address this, we enforce atomicity at the programming level by wrapping `relay` statements in a `try` block and inserting a `catch` block that invokes a programmer-defined `revert()` function. For key datatypes, such as digital asset types with Move semantics [13], the Crystality transpiler provides default `revert()` implementations, ensuring objects are returned back to their original accounts.

4 Implementations of Compilation

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 `@global`, `@engine`, or a name of Solidity elementary typename like `@address` or `@uint`. `@global` can be omitted as the default κ-scope specifier. A variable definition with `@global` or `@engine` is converted to Solidity simply as:

```
var_type var_name;
```

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

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

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

with the scope specifier recorded in the symbol table for compatibility check when 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 specifier. In cases of `@global` or `@engine`, the declaration is 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 an argument to allow accessing variables defined in the current keyed κ-scope.

Code in a function body has immediate access to variables and functions in the κ-scope by referring corresponding symbols. Symbols defined in the `@global` and `@engine` κ-scopes are merged with ones in the κ-scope without scope qualifications. Symbols defined in any κ-scope should have unique names within a smart contract. In the κ-scope ϕ_k or current engine κ-scope $\phi_{\theta(i)}$, variables and functions defined in global scope are merged for read-only access. Symbols defined in current engine κ-scope $\phi_{\theta(i)}$ are merged into the target κ-scope ϕ_k, allowing full access of both read and write.

4.3 Relay to a Target κ-Scope

To continue execution in a different κ-scope other than the current κ-scope, 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 followed by the target specifier which can only be the `@global`, a specific `key` of keyed κ-scopes or an expression resulting in 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 by the EVM host. So that, it can be captured and reinterpreted as a cross-scope relay transaction. The transpiler-converted Solidity code is shown as follows:

```
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, ...
  )
);
```

, where `_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 a relay transaction is received, a private function in the target κ-scope is invoked. It is also transpiler-generated by taking the body of the lambda function as:

```
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.

4.4 Transpilation for EVM

We adopt a two-stage process to compile Crystality smart contracts into EVM bytecodes. Initially, our transpiler converts Crystality code to Solidity code. Utilizing ANTLR [1], we generate the parser code based on Solidity syntax definitions extended with Crystality directives. The resulting abstract syntax tree (AST) is traversed to produce Solidity code. Subsequently, the generated Solidity code is compiled by the SOLC (version 0.8.18)[4] compiler, resulting in EVM bytecodes executable on an unmodified EVM. The `ERC20` contract in Figure 2 is 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 (version 0.8.0)[18] for executing SOLC-generated bytecode. EVMC[3], the low-level ABI between EVMs and Ethereum clients, serves as the standard for interacting with EVMOne. We implement its `HostContext` interface to provide EVMOne with capabilities of accessing state storage, handling message call, and exposing metadata of current block and transaction. All λ-relay invocations are captured by overriding the `HostContext::call` function, using the magic address `_magic_address_kappa` and `_magic_address_global`. The target scope, function, and arguments, as encoded by `abi.encodeWithSignature`, are passed to the underlying system for composing the relay and forwarded to the destination EVM of the target κ-scope.

4.5 Discussions

Practical Experiences: To efficiently use Crystality, we recommend three key strategies. First, partition contract states by using the `@address` directive to distribute contract states across VMs, enabling parallel transaction execution, as in the token transfer example. Second, localize global state access by leveraging `@engine` to process global states within individual VMs and deferring global updates until all transactions are completed, as in the voting example. Third, minimize relays to global states to reduce synchronization overhead between VMs. These practices promote concurrent state processing while limiting dependencies on global states.

Program Rewriting: A limitation of Crystality is the need for developers to rewrite contracts using its directives. However, automating the translation from Solidity to Crystality is achievable. For example, Solidity's `mapping` keyword, often used to link states to addresses (e.g., `"mapping(address => uint256) balances"` in ERC20), can be directly mapped to `@address` scopes, while non-`mapping` states could default to `"@global"`. Additionally, the transpiler could analyze Solidity code, detect state transitions, and insert Crystality's `relay` directives, encapsulating the state-changed code region in lambda functions. This approach would allow existing Solidity contracts to take advantage of Crystality's execution engine with minimal developer effort.

This paper emphasizes manual rewriting for two reasons. First, it enables customized parallel algorithms for specific contracts. For example, voting contracts can leverage local states within VMs for initial voting, followed by result aggregation. Second, automating translation requires substantial engineering effort. Future efforts could develop tools to automate the translation of routine Solidity contracts into Crystality-optimized code, reserving manual intervention for specified parallelism when necessary.

5 Evaluation

In our evaluation, alongside `MyToken` (ERC20), we incorporate four widely used smart contracts in Crystality: `Voting`[6], `AirDrop`[2], `CryptoKitties`[36], and `MillionPixel`[5]. The contracts are deployed and executed on Ethereum.

```
contract Ballot{
  Proposal[] proposals;

  function vote(uint proposal) public {
    proposals[proposal].voteCount += 1;
  }
}
```

(a) in Solidity

```
contract Ballot{
  Proposal[] @global proposals;
  Proposal[] @engine ps_engine;

  function vote(uint proposal)
    @address public {
    ps_engine[proposal].voteCount += 1;
  }

  function finalize() @global public {
    relay @engines () {
      relay @global (ps_engine) {
        for(int i=0; i<ps_engine.len; i++)
            proposals[i] += ps_engine[i];
      }
    }
  }
}
```

(b) in Crystality

```
// Aptos contract
module 0x1::Ballot{
    fun initialize(forums:vector<address>,proposals:vector<Proposal>){
        for(forum in forums){ //Init candidates...});
    }

    fun vote(forum:address, proposal:u64) {
      borrow_global_mut<Proposals>(forum)[proposal].voteCount+= 1;
    }

    fun finalize(forums: vector<address>):vector<u64> {
      for(forum in forums){ //Collect all votes for the final result }
}

// Aptos client
// step 0. Init #_of_addresses as #_of_shards for voting pools(forum)
let forums :vector<address> = vec![<addr_0>,<addr_1>...<addr_n-1>];
// step 1. Initialize candidates for every forum
for i in 0..shards {
  invoke_contract<Ballot.initialize>(forums[i],proposals);
}
// step2. Voters goes to forums based on addresses
let index = sender.address().to_bigint() % shards;
invoke_contract<Ballot.vote>(forums[index].address(), proposal_index);
//step3. Finalize result
let res = invoke_contract<Ballot.finalize>(forums);
```

(c) in Aptos

Figure 6. A glance at the `Voting` smart contract in Solidity, Crystality, and Aptos Move, respectively.

We use the `Voting` contract as an example to show how to use Crystality to express the parallel algorithm. The `Voting` contract collects voting results for candidates, traditionally using a shared candidate array for sequential voting. A parallel approach involves creating a temporary array on each parallel unit (i.e., VM), enabling concurrent voting, followed by aggregating results from these temporary arrays.

Figure 6 compares the implementations in Solidity, Crystality, and Aptos Move. In Crystality, the `@engine` directive introduces the `ps_engine` variable for parallel reads/writes in each VM, and a `finalize()` function for handling the result aggregation by relaying between global and engine scopes. The figure shows that Crystality simplifies the parallel `Voting` implementation, whereas Aptos requires more complex contract and client-side steps, such as creating multiple forums and managing end-users to vote across forums.

State access patterns also affect performance, besides state access dependencies. **`MyToken`** follows a one-to-one state access pattern, while **`Voting`** employs a many-to-one pattern. **`Airdrop`** employs a one-to-many update pattern, and **`CryptoKitties`** features chained state access. **`MillionPixel`** uses a one-to-one model but partitions states by coordinates rather than addresses. These examples reflect typical patterns in practical smart contracts. With Crystality's state partitioning, state-level parallelism is automated through its execution engine in a pipelined manner.

5.1 Experimental Setups

We evaluate five contracts on a 64-core AMD EPYC 7742 (3.4GHz) CPU with 2TB memory, running Linux Ubuntu 20.04. We use the latest open-source code of Aptos and Sui from their GitHub repos, configure them in the single-node benchmark crates, and optimize performance to the best of our ability. For comparisons between Crystality and Solidity/Ethereum, we exclude overheads like transaction signature verification. When comparing Crystality to Aptos and Sui, we employ similar setups for all systems to ensure fair comparisons, excluding consensus overhead but including signature verification.

We use randomly generated workloads for all contracts, including 10,000 addresses and 100,000 transactions for Aptos, Sui, and Crystality. For `MyToken`, we also replay ETH historical transactions. Utilizing the Etherscan API [8], we obtain ETH transfer transactions in January 2024. We prepare batches of every 100,000 successive transactions. Due to page limits, we only show the results of utilizing the first batch, including those transactions from block heights 18,908,895 to 18,910,315. After converted to the proper formats, the dataset is replayed on Aptos, Sui, and Crystality.

5.2 Compared to Solidity on EVMs

Figure 7 shows the TPS of Crystality on the 64-core machine, running 1,000,000 or 100,000 randomly generated transactions for these contracts, excluding overhead like signature verification. We observed a 30.3x to 56.1x TPS improvement with 64 EVMs compared to a single EVM. For reference, we included performance metrics for Solidity contracts, which transactions are executed sequentially. Thus, multiple EVMs do not improve their throughput and the TPS for Solidity contracts is shown using one EVM, denoted by "Eth-" prefix. Compared to Solidity, Crystality achieves a 17x to 42.9x TPS improvement when run on 64 EVMs.

On a single EVM, Crystality contracts show lower TPS compared to their Solidity counterparts. This discrepancy stems from the relay implementation in the transpiler-generated code, as detailed in Section 4.4. The relay mechanism employs the `address.call()` method, forwarding execution to the EVM hosting the target κ-scope and re-entering the EVM for relay processing. EVMOne execution involves two steps: `analyze()` (code review and bytecode parsing) and `execute()` (function lookup and instruction execution). Figure 8 shows that in most cases both steps take longer for transpiler-generated code due to larger bytecode sizes and additional instructions required for relays.

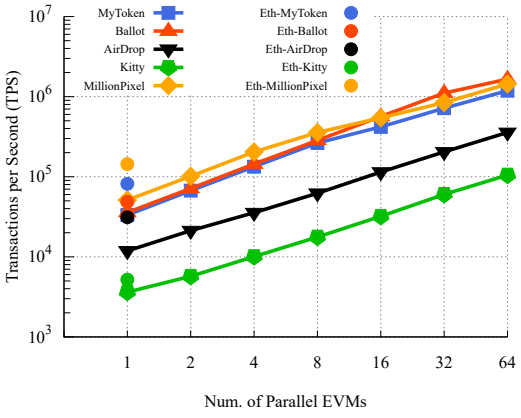

Figure 7. TPS of Crystality's contracts on a 64-core machine.

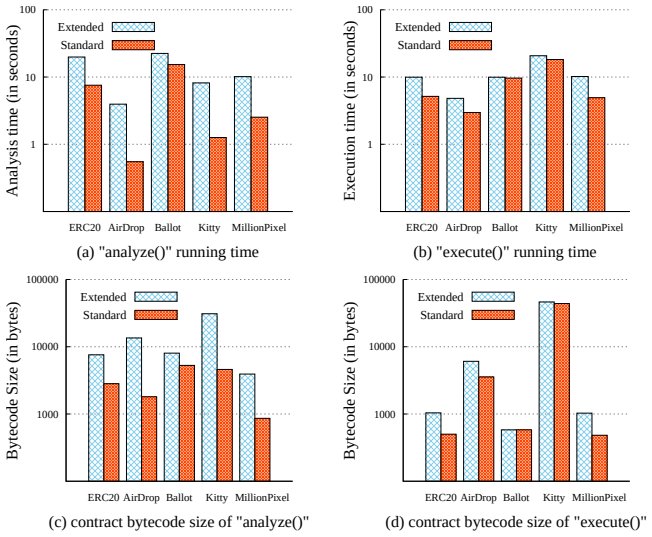

Figure 8. Accumulated execution time in "analyze()" and "execute()" functions, and respective bytecode sizes of smart contracts.

Figure 8(d) shows the standard and extended `Voting` contracts have identical bytecode size in the `execute()` function. This is because the experiment only measured transactions invoking the `vote()` function, which is identical in both versions. The `relay` is called only in the `finalize()` function, as shown in Figure6.

5.3 Compared to Aptos and Sui

Table 1 presents the average execution time per transaction on a single VM for all contracts. Given the different programming languages and VMs used by Aptos, Sui, and Crystality, we use relative speedup, calculating the speedup on multiple VMs over each system's baseline on one VM.

Table 1. Average transaction execution latency (in us) on one VM

	MyToken (hist.)	MyToken (rand.)	Ballot	AirDrop	CryptoKitties	MillionPixel
Aptos	413.18	405.19	256.69	359.57	1,679.24	274.84
Sui	619.41	549.6	593.7	904.1	1,949.9	774
Crystality	148.72	149.06	144.07	207.9	383.4	138.98

Figure 9 displays the relative speedups of Aptos, Sui and Crystality on the 64-core machine. For `MyToken` contract with historical and random transactions, Aptos achieves up to 5.57x and 19.13x speedups compared to its single VM baseline. These variations are explained by Aptos's profiling data in Table 2 and Table 3. Aptos transaction execution involves execution and validation, with a time-consuming `SUSPEND` status that holds a transaction until it can be resumed. For random transactions on 64 VMs, execution and validation totals are 102,219 and 139,426; for historical transactions, these totals rise to 186,948 and 667,148, with transaction suspensions increasing from 66 to 46,913. Given the average 413-microsecond sequential execution time for an Aptos `MyToken` transaction, the 6,231.35-microsecond average suspend time significantly reduces speedup.

When running `MyToken` on 64 VMs, Sui achieves up to 12.47x and 13.74x speedups over its baselines, while Crystality achieves 38.07x and 47.01x speedups for historical and random transactions, respectively. Sui's `MyToken` transactions avoid conflicts by operating on owned objects, eliminating the overhead of state dependency analysis and locking.

For the `Voting` contract with random transactions, Aptos, Sui, and Crystality achieve 10.68x, 12.1x, and 53.36x speedups on 64 VMs compared to their own baselines. For the `AirDrop`, `CryptoKitties`, and `MillionPixel` contracts, Crystality achieves speedups of 38.89x, 34.59x, 47.31x, while Aptos shows 19.46x, 3.29x, 11.51x speedups, and Sui gets 15.72x, 20.22x, 15.64x speedups over their own baselines.

Figure 9 shows that Sui outperforms Crystality in scalability for `CryptoKitties` on 4-32 cores. However, Crystality achieves higher absolute throughput than Sui across all core configurations. Since the speedup comparison normalizes throughput based on each system's single-core performance, Sui's better scalability on 4-32 cores is partly due to its lower single-core throughput. Additionally, Sui's implementation of `CryptoKitties` uses owned objects for kitties, avoiding shared-object contention, a design not provided in Aptos. Even when using Aptos's efficient `0x1::smart_vector` for `CryptoKitties`, its performance scalability remains limited.

Figure 10 shows the execution time breakdown for Sui across 64 VMs using 1,000 randomly selected transactions

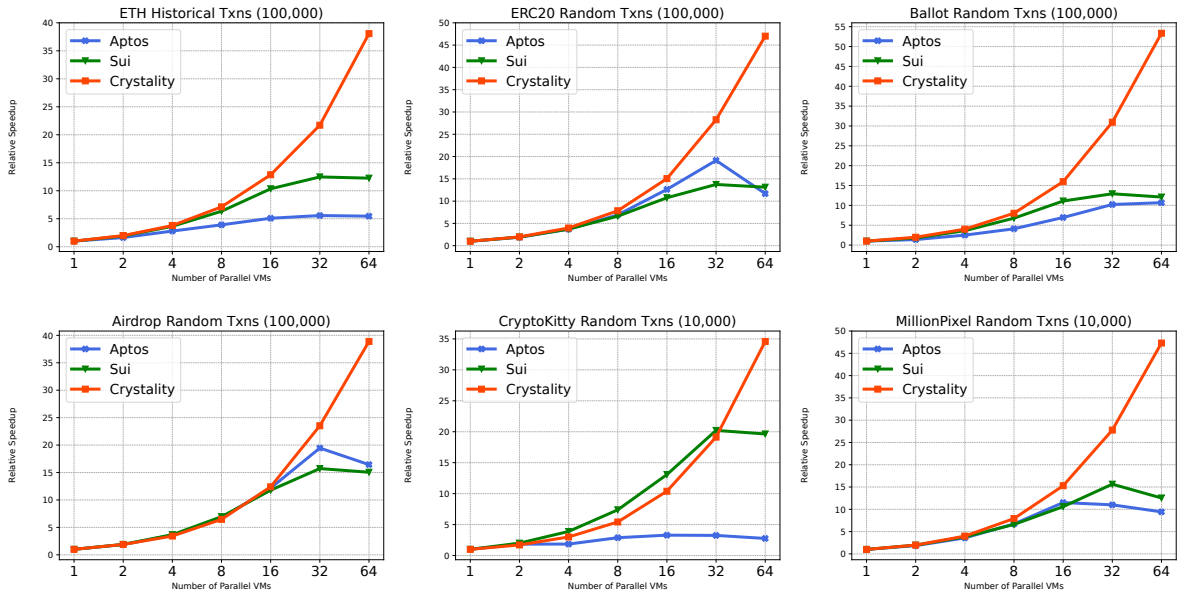

Figure 9. Relative Speedups for Aptos, Sui, and Crystality over their own baselines.

Table 2. Profiling data of Aptos `MyToken` (historical) on VMs

	1	2	4	8	16	32	64
num._exec.	100,000	116,262	128,252	140,472	154,410	169,682	186,948
num._val.	100,000	123,590	154,002	210,276	312,735	481,654	667,148
num._abot.	0	16,262	28,252	40,472	54,410	69,682	86,948
num._susp.	0	6,250	20,804	29,734	38,657	45,572	46,913
avg._susp._time(us)	0	383.18	291.42	696.25	1451.69	2949.63	6231.35

Table 3. Profiling data of Aptos `MyToken` (random) on VMs

	1	2	4	8	16	32	64
num._exec.	100,000	100,037	100,085	100,221	100,442	101,847	102,219
num._val.	100,000	100,062	100,179	101,860	102,980	110,477	139,426
num._abot.	0	37	85	221	442	1847	2219
num._susp.	0	0	1	0	4	19	66
avg._susp._time(us)	0	0	241.73	0	338.19	430.34	674.57

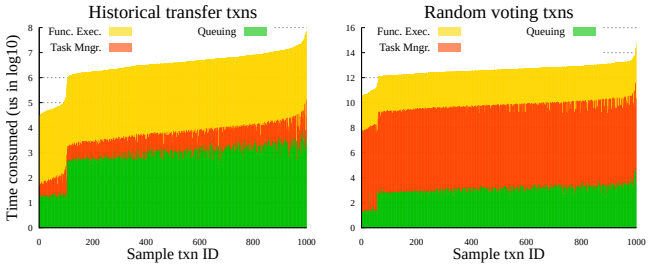

Figure 10. Execution time breakdown for Sui on 64 VMs.

from 100,000. Sui's three stages are: (1) Queuing Time – waiting for transaction manager processing; (2) Task Management Time – from being placed in the `ExecutingTxns` or `PendingTxns` hashmap to being processed by `Execution Driver`; and (3) Function Execution Time – contract execution time in a worker. For token transfers, task management time is much shorter than others, since Sui's token transfer contract uses owned objects and concurrency control is bypassed. In contrast, for `Voting`, shared object locking causes task management time to dominate, exceeding queuing and function execution by 2-4 orders of magnitude.

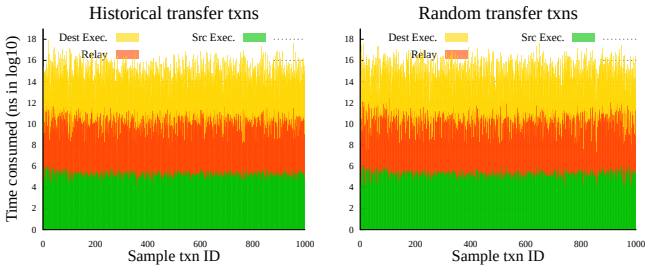

Figure 11. Execution time breakdown for Crystality on 64 EVMs.

Figure 11 presents Crystality's execution time on 64 EVMs for 100,000 historical ETH and random transfer transactions. For 1,000 samples, times ([min, max, avg], in nanoseconds) are: ETH – Sender processing: [9,829, 1,685,167, 221,144]; Relaying: [1,233, 1,919,952, 180,080]; Receiver processing: [4,598, 1,080,860, 89,553]; Random – Sender processing: [10,119, 1,668,486, 254,245]; Relaying: [1,272, 1,322,980, 194,312]; Receiver processing: [6,052, 1,463,226, 167,399]. The results indicate that relay times remain consistently lower than state processing time, ensuring efficient overlap.

6 Related Work

Compared to conventional parallel models such as OpenMP [11], STM [47], and PGAS [16], Crystality stands out by addressing three core aspects: transactions (input), states (data), and VMs (parallel units). While it shares data partitioning principles with PGAS, Crystality focuses on transaction processing. Unlike PGAS, which handles entire datasets (e.g., vectors or matrices), Crystality processes partitioned data

specific to transactions, addressing transactional requirements. In contrast, STM-based models like Block-STM lack explicit state partitioning. Crystality integrates PGAS and STM concepts with blockchain-specific innovations, such as MPT trees for result verification, consensus-layer transaction ordering for deterministic updates, and leveraging smart contract commutativity. Crystality's transpiler converts its directives to Solidity, ensuring EVM compatibility without altering EVM opcodes or relying on external Solidity events. These features simplify adoption in the blockchain community, distinguishing Crystality from traditional models.

Several studies improve blockchain performance through optimized transaction/block propagation [19, 25, 39, 40, 43], accelerated consensus protocols [24, 29, 33, 37, 50, 59], and Layer 2 networks [44, 48, 49]. Two related directions are:

Parallel VMs: Dickerson et al. [26] employ transactional boosting to enable miners to execute conflict-free transactions in parallel. Anjana et al. [10] use optimistic STM for concurrent execution on miners and verification on full nodes. Saraph and Herlihy [46] assess the benefits of speculative execution using historical Ethereum data. Chen et al. [14] propose Forerunner, a constraint-based speculative model. Gelashvili et al. [23] combine STM with OCC in Block-STM, the engine behind Aptos. Kniep et al. [27] extend parallel VM execution from single machines to distributed environments.

Multi-chain and Sharding: Elastico [35] implements sharding by assigning nodes to committees via PoW puzzles, with PBFT for intra-shard consensus and a final committee aggregating results. OmniLedger [28] uses RandHound [51] for validator selection and Atomix for cross-shard transactions. RapidChain [62] optimizes intra-committee consensus. Monoxide [57] uses PoW for cross-shard processing and merge-mining to prevent 1% attacks. OHIE [60] uses parallel PoW instances secured by a Merkle tree. Prism [12] decomposes blockchains into specialized chains, with an extension [56] for contract execution. Chainspace [9] and COSPLIT [42] provide parallel execution in sharded systems, using optimistic methods or state dependency analysis. Saber [34] separates consensus and execution nodes, enabling parallel execution and asynchronous verification, respectively.

7 Conclusions

The sequential design of smart contract languages like Solidity and MOVE limits the potential of parallel VMs, while current optimistic and pessimistic methods struggle to efficiently handle asynchronous execution for commutative transactions. This paper presents Crystality, a parallel programming model for smart contracts on parallel EVMs, which simplifies expressing parallelism through Programmable Contract Scopes and Asynchronous Functional Relay. As a Solidity extension, Crystality transpiles code into standard Solidity to ensure EVM compatibility. Its underlying system supports asynchronous execution for commutative transactions and employs an optimistic fallback to preserve transaction order. Our experiments on a 64-core machine demonstrate that Crystality outperforms Ethereum, Aptos, and Sui, enabling more scalable and efficient decentralized applications.

8 Acknowledgments

This work is supported in part by the National Key R&D Program of China under Grant 2022YFB2702201. We thank the reviewers for their invaluable feadback and comments.

References

[1] [n. d.]. ANother Tool for Language Recognition. https://www.antlr.org/.
[2] 2018. Solidity AirDrop Smart Contract. https://github.com/SpringRole/smart-contracts/blob/master/contracts/AirDrop.sol.
[3] 2022. Ehtereum Client-VM Connector API. https://github.com/ethereum/evmc.
[4] 2023. Installing the Solidity Compiler. https://github.com/ethereum/solidity/blob/develop/docs/installing-solidity.rst.
[5] 2023. Million Pixel DFI. https://millionpixeldfi.github.io.
[6] 2023. Solidity by Example: Voting. https://docs.soliditylang.org/en/v0.8.21/solidity-by-example.html.
[7] 2024. BTCScan: Bitcoin Blockchain Explorer. https://btcscan.org/.
[8] 2024. The Ethereum Blockchain Explorer. https://etherscan.io/.
[9] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A sharded smart contracts platform. In *NDSS '18*.
[10] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts. In *PDP '19*. 83–92.
[11] The OpenMP ARB. 2024. OpenMP Application Programming Interface (Version 6.0). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf.
[12] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the Blockchain to Approach Physical Limits. In *CCS '19*. 585–602.
[13] 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. 2020. Move: A Language With Programmable Resources. https://diem-developers-components.netlify.app/papers/diem-move-a-language-with-\programmable-resources/2020-05-26.pdf.
[14] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-Based Speculative Transaction Execution for Ethereum. In *SOSP '21*. 570–587.
[15] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2013. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP '13'*. 1–17.
[16] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *ACM Comput. Surv.* 47, 4, Article 62 (May 2015), 27 pages.
[17] Ethereum. 2018. Ethereum Merkle Patricia Trie (Extension node). https://ethereum.stackexchange.com/questions/39915/ethereum-merkle-patricia-trie-extension-node.
[18] The Ipsilon (ex Ewasm) team. 2022. evmone: Fast Ethereum Virtual Machine implementation. https://github.com/ethereum/evmone.
[19] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI'16*. 45–59.
[20] Aptos Foundation. 2022. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure. https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf.

[21] Solana Foundation. 2019. 8 Innovations that Make Solana the First Web-Scale Blockchain. https://solana.com/news/8-innovations-that-make-solana-the-first-web-scale-blockchain.

[22] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. 2022. Utilizing Parallelism in Smart Contracts on Decentralized Blockchains by Taming Application-Inherent Conflicts. In *ICSE '22*. 2315–2326.

[23] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *PPoPP '23*. 232–244.

[24] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *SOSP '17*. 51–68.

[25] Yilin Han, Chenxing Li, Peilun Li, Ming Wu, Dong Zhou, and Fan Long. 2020. Shrec: Bandwidth-Efficient Transaction Relay in High-Throughput Blockchain Systems. In *SoCC '20*. 238–252.

[26] Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *PPoPP '08*. 207–216.

[27] Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. 2024. Pilotfish: Distributed Transaction Execution for Lazy Blockchains. *arXiv preprint arXiv:2401.16292* (2024).

[28] Eleftherios Kokoris Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *SP '18*. 583–598.

[29] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *SEC'16*. 279–296.

[30] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: multi-data center consistency. In *EuroSys '13*. 113–126.

[31] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* (1981).

[32] Sei Labs. 2023. Sei: The Layer 1 for Trading. https://github.com/sei-protocol/sei-chain/blob/main/whitepaper/Sei_Whitepaper.pdf.

[33] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *USENIX ATC '20*. 515–528.

[34] Jian Liu, Peilun Li, Raymond Cheng, N. Asokan, and Dawn Song. 2022. Parallel and Asynchronous Smart Contract Execution. *IEEE Trans. Parallel Distrib. Syst.* 33, 5 (may 2022), 1097–1108.

[35] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *CCS '16*. 17–30.

[36] Arpit Mathur. 2017. Cryptokitties Contract from the Eth blockchain. https://gist.github.com/arpit/071e54b95a81d13cb29681407680794f.

[37] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *CCS '16*. 31–42.

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

[39] Gleb Naumenko, Gregory Maxwell, Pieter Wuille, Alexandra Fedorova, and Ivan Beschastnikh. 2019. Erlay: Efficient Transaction Relay for Bitcoin. In *CCS '19*. 817–831.

[40] A. Pinar Ozisik, Gavin Andresen, Brian N. Levine, Darren Tapp, George Bissias, and Sunny Katkuri. 2019. Graphene: Efficient Interactive Set Reconciliation Applied to Blockchain Propagation. In *SIGCOMM '19*. 303–317.

[41] Monad Pad. 2024. The Monad White Paper. https://files.monadpad.xyz/whitepaper.pdf.

[42] George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical Smart Contract Sharding with Ownership and Commutativity Analysis. In *PLDI '21*. 1327–1341.

[43] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. 2021. RainBlock: Faster Transaction Processing in Public Blockchains. In *USENIX ATC '21*. 333–347.

[44] Joseph Poon and Thaddeus Dryja. 2016. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf.

[45] Xiaodong Qi, Jiao Jiao, and Yi Li. 2023. Smart Contract Parallel Execution with Fine-Grained State Accesses. In *ICDCS '23*. 841–852.

[46] Vikram Saraph and Maurice Herlihy. 2019. An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts. In *Tokenomics '19*.

[47] Nir Shavit and Dan Touitou. 1995. Software transactional memory. In *PODC '95*. 204–213.

[48] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Mohammad Alizadeh, Giulia Fanti, and Pramod Viswanath. 2018. Routing Cryptocurrency with the Spider Network. In *HotNets '18*. 29–35.

[49] Vibhaalakshmi Sivaraman, Shaileshh Bojja Venkatakrishnan, Kathleen Ruan, Parimarjan Negi, Lei Yang, Radhika Mittal, Giulia Fanti, and Mohammad Alizadeh. 2020. High Throughput Cryptocurrency Routing in Payment Channel Networks. In *NSDI '20*. 777–796.

[50] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. 2016. SPECTRE: Serialization of Proof-of-work Events: Confirming Transactions via Recursive Elections. https://eprint.iacr.org/2016/1159.pdf.

[51] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. 2017. Scalable bias-resistant distributed randomness. In *SP '17*. 444–460.

[52] Ethereum Dev Team. 2021. Solidity Documentation. https://docs.soliditylang.org/en/latest/.

[53] Ethereum Dev Team. 2022. Ethereum Virtual Machine (EVM). https://ethereum.org/en/developers/docs/evm/.

[54] The MystanLabs Team. 2023. The Sui Smart Contracts Platform. https://docs.sui.io/paper/sui.pdf.

[55] The NEAR Team. 2022. The NEAR White Paper. https://near.org/papers/the-official-near-white-paper/.

[56] Gerui Wang, Shuo Wang, Vivek Bagaria, David Tse, and Pramod Viswanath. 2020. Prism removes consensus bottleneck for smart contracts. In *CVCBT '20*. 68–77.

[57] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale Out Blockchain with Asynchronous Consensus Zones. In *NSDI '19*. 95–112.

[58] Gavin Wood. 2016. Polkadot: Vision for a Heterogeneous Multi-Chain Framework. https://assets.polkadot.network/Polkadot-whitepaper.pdf.

[59] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC '19*. 347–356.

[60] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain scaling made simple. In *SP '20*. 90–105.

[61] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08*. 1–14.

[62] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *CCS '18*. 931–948.

A Artifact Evaluation

A.1 Abstract

This artifact includes the source code, datasets, and scripts required to reproduce the results presented in the paper. It is available on Zenodo: https://zenodo.org/records/14534549.

A.2 Getting Started Guide

A.2.1 Description.

- **Test Suites:** All code for Crystality, as well as modified codebases for Aptos and Sui to replay historical transactions, are included. The scripts for building and running these systems are also contained. Aptos and Sui will be downloaded from their GitHub repositories by running the `setup.sh` script.
- **Aptos and Sui:** This artifact uses:
  - **Aptos:** at commit ID: **a182...7b0b**.
  - **Sui:** at commit ID: **4dbc...c9c0**.

A.2.2 Requirements.

- **Processor:** Intel Core i9-13900K or AMD EPYC 7742 are recommended. Use only performance cores on Intel CPUs and only physical cores (not logical threads with HT or SMT).
- **Memory:** 32 GB or higher is recommended.
- **Operating System:** Ubuntu 20.04.
- **Dependencies:** openssl 3.0.15; pip3; Cmake 3.16.3; llvm; lld; clang 12.0,1; curl; git; rustc 1.78.0; python 3.10; gnuplot 6.0; bc.
- **Compilation:** g++11.

A.2.3 Setup Instructions.

- Set up the environment:

```
$ export TEST_HOME=/path/to/crystality-test-suites
$ cd $TEST_HOME/scripts
$ ./setup.sh
```

- Run all benchmarks:

```
$ cd $TEST_HOME/scripts/
$ ./run_benchmarks.sh
```

This script runs all experiments and generates figures and tables. Results (raw data) are saved in the **$TEST_HOME/results/**. Figures and tables are saved in **$TEST_HOME/pictures/**.

A.3 Step-by-Step Instructions

A.3.1 TPS of Crystality's Contracts. This script measures the TPS of Crystality smart contracts and their Solidity equivalents (Figure 7). Results are saved in **$TEST_HOME/repos/crystality/gnuplot/performance.csv** and **$TEST_HOME/pictures/figure7/**.

```
$ cd $TEST_HOME/scripts/
$ ./crystality_performance.sh
```

A.3.2 "analyze()" and "execute()". This script collects contract bytecode sizes and runtimes for the `analyze()` and `execute()` EVM functions (Figure 8). Results are saved in **$TEST_HOME/repos/crystality/gnuplot/execution_breakdown.csv** and **$TEST_HOME/pictures/figure8/**.

```
$ cd $TEST_HOME/scripts/
$ ./crystality_execute_analyze.sh
```

A.3.3 Average Transaction Execution Latency. This script measures the average transaction execution latency for Aptos, Sui, and Crystality (Table 1). Results are saved in **$TEST_HOME/pictures/table1**.

```
$ cd $TEST_HOME/scripts/
$ ./transaction_latency_comparison.sh
```

A.3.4 Profiling Data of Aptos MyToken Transactions. This script profiles Aptos transactions (historical and random), generating data for Tables 2 and 3. Results are in **replay_results.log** and **random_results.log** in **$TEST_HOME/repos/aptos-core/aptos-move/aptos-transaction-benchmarks/scripts/**. Tables are in **$TEST_HOME/pictures/** as `table2` and `table3`.

```
$ cd $TEST_HOME/scripts/
$ ./aptos_profiling.sh
```

A.3.5 Speedup Comparison. This script compares the scalability of Aptos, Sui, and Crystality using their respective TPS on one VM as a baseline (Figure 9). Results are saved in **$TEST_HOME/results/** and **$TEST_HOME/pictures/figure9/**.

```
$ cd $TEST_HOME/scripts/
$ ./speedup_comparison.sh all
```

A.3.6 Execution Time Breakdown for Sui. This script collects Sui's execution time breakdown for historical and random transactions (Figure 10). Results are saved in **erc20_analyze.csv** and **ballot_analyze.csv** in **$TEST_HOME/repos/sui/crates/sui-single-node-benchmark/gnuplot/**. Figures are saved in **$TEST_HOME/pictures/figure10/**.

```
$ cd $TEST_HOME/scripts/
$ ./sui_analyze.sh
```

A.3.7 Execution Time Breakdown for Crystality . This script analyzes Crystality's execution time breakdown, highlighting relay overhead (Figure 11). Results are saved in **time_info_random.csv** and **time_info_replay.csv** under **$TEST_HOME/repos/crystality/**. Figures are saved in **$TEST_HOME/pictures/figure11/**.

```
$ cd $TEST_HOME/scripts/
$ ./crystality_execution_breakdown.sh
```