Practical Smart Contract Sharding with Ownership and Commutativity Analysis

Anonymous Author(s)

Abstract

Sharding is a popular way to achieve scalability in blockchain protocols, increasing their throughput by partitioning the set of transaction validators into a number of smaller committees, splitting the workload. Existing approaches for blockchain sharding, however, do not scale well when concurrent transactions alter the *same* replicated state component—a common scenario in Ethereum-style smart contracts.

We propose a novel approach for efficiently sharding such transactions. It is based on a folklore idea: state-manipulating atomic operations that *commute* can be processed in parallel, with their cumulative result defined deterministically, while executing non-commuting operations requires one to *own* the state they alter. We present CoSplit—a static program analysis tool that soundly infers *ownership and commutativity summaries* for arbitrary smart contracts written in the Scilla language and translates those summaries to *sharding signatures* that are used by the blockchain protocol to allocate shards to transactions in a way that maximises parallelism. We have integrated CoSplit into a state-of-the-art sharded blockchain protocol. Our evaluation shows that using CoSplit introduces negligible overhead to the transaction validation cost, while the inferred sharding signatures allow the system to achieve a significant increase in transaction processing throughput for real-world smart contracts.

1 Introduction

The idea of Nakamoto consensus (*aka* blockchain) has been instrumental for enabling decentralised digital currencies, such as Bitcoin [51]. The applications of blockchains have further expanded with the wide-spread adoption of *smart contracts* [65]—self-enforcing, self-executing protocols governing an interaction between several mutually distrusting parties. The Ethereum blockchain has provided a versatile framework for defining smart contracts as blockchain-replicated stateful objects identified by their account numbers [69].

The open and decentralised nature of Nakamoto consensus comes at the price of throughput scalability. At a high level, in order for a sequence of transactions (so-called *block*) to be agreed upon system-wide, the system's participants (so-called *miners*) have to validate those transactions, with each miner executing them individually [4]. As a result, the throughput of blockchain systems such as Bitcoin and Ethereum does not improve, and even slightly deteriorates, as more participants join the system: Bitcoin currently processes up to 7 transactions per second [7], while Ethereum's throughput is 11.3 transactions per second. Even worse, popular smart contracts may cause high congestion, forcing protocol participants to exclusively process transactions specific to those contracts. This phenomenon has been frequent in Ethereum: in the past, multiple ICOs (Initial Coin Offering, a form of a crowdfunding contract) and games, such as CryptoKitties, have rendered the system useless for any other purposes for noticeable periods of time [15].

Sharding in blockchains. One of the most promising approaches to increase blockchain throughput is to split the set of miners into a number of smaller committees, so they can process incoming transactions in parallel, subsequently achieving a global agreement via an additional consensus mechanism—an idea known as *sharding*. Sharding transaction executions, as well as sharding the replicated state, has been an active research topic recently, both in industry [25, 32, 42, 54, 62, 68, 72] and academia [1, 18, 39, 48, 71].

Many of those works focus exclusively on sharding the simplest kind of transactions—*user-to-user transfers of digital funds,*—which are paramount in blockchain-based cryptocurrencies, while ignoring sharding of smart contracts [39, 48, 71, 72]. Existing proposals tackling smart contracts impose heavy restrictions on contract-manipulating transactions, for instance, requiring the accounts of both the contract and its user to be assigned the same shard, or processing all such transactions in a specialised shard [18, 32, 42]. Other solutions assume a complex cross-shard communication protocol to reconcile possible conflicts [25, 32, 54, 62, 68], or adopt a contract design very different from Ethereum [1].

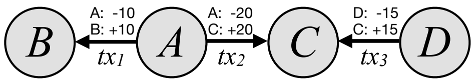
To the best of our knowledge, none of these approaches allows for parallel sharded executions involving the *same* smart contract. That is, none solve the mentioned congestion problem in Ethereum, caused by highly-popular contracts.

In this work, we describe a novel approach for significantly increasing the throughput of blockchains for smart contract-manipulating transactions. To achieve this, instead of treating contract implementations as "black boxes" (as do all the works mentioned above), we design a solution based on PL techniques, specifically, on static program analysis.

Our approach. Why can user-to-user money transfers be sharded efficiently without complex inter-shard communication, and how can we generalise (perhaps, conservatively) this logic to shard arbitrary smart contracts?

PL'18, January 01–03, 2018, New York, NY, USA
2018.

Consider a transaction tx_1 that manifests a transfer of 10 units of some digital currency from the user A to B, and a transaction tx_2 that states that A transfers 20 units to C. In order to ensure that A does not double-spend, both tx_1 and tx_2 have to be executed in the same shard—the one that ***owns*** A's account and keeps track of A's balance. However, neither B nor C need to be owned by A's shard: as long as tx_1 and tx_2 are validated within A's shard, the positive deltas to B and C's accounts can be simply broadcast through the network, so their balances are increased accordingly with no extra inter-shard interaction.



Now consider a transaction tx_3, in which D transfers 15 units to C. Notice that it does not matter in which order tx_1 and tx_3 are going to be processed, as they ***commute***: either of their relative orderings will increase C's balance by 35.

The notions of state *ownership* and operation *commutativity* have been central in a number of works dedicated to reasoning about deterministic parallelism and proving correctness of concurrent programs [21, 22, 24, 37, 43, 47, 53, 70]. In those works, the ownership discipline determines what parts of the shared state need to be manipulated sequentially by the *same* thread, while commutativity allows certain actions to be executed in *concurrent* threads in parallel, with a deterministic result. The virtues of commutative operations have also been studied in the systems community for scaling concurrent software [3, 14, 56, 58] and achieving faster consensus in replication protocols [12, 40, 45, 50]. However, to the best of our knowledge, no attempts to *automatically* leverage commutativity in *user-defined replicated computations* (*e.g.*, smart contracts) have been made to date.

In this work, we present CoSplit—a static analysis tool that soundly infers both *ownership* and *commutativity* information from source code of smart contracts and translates it to *sharding signatures*. The signatures are used, upon the deployment of a contract, to define a *sharding strategy* for the contract-manipulating transactions via the following rules:

- All transactions touching parts of a contract's state *owned* by a shard $\mathcal{S}$ must be executed in this shard;
- Transactions executed in different shards are guaranteed to *commute*. Their cumulative result can be obtained by means of "joining" their respective contributions in a way prescribed by the sharding signature.

These two rules allow the system to enjoy a notion of consistency for parallel transaction executions adopted from works on the semantics of concurrent revisions [8, 9, 46]:

1. Potentially conflicting contract-manipulating transactions will be executed in some *globally-agreed order*.
2. Commuting transactions can be executed in parallel, as their effect *does not depend* on their order.

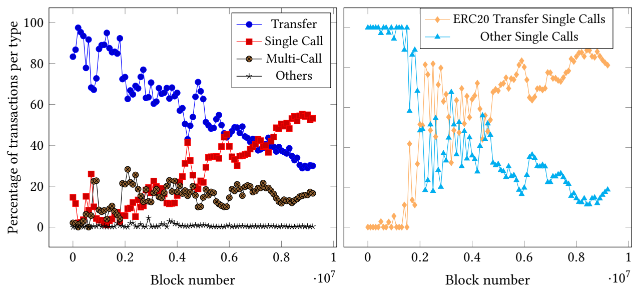


Figure 1. Left: Ethereum transaction breakdown per type; the percentage distribution is averaged over 100K block periods. Right: breakdown of single-call transactions.

As we will discuss in Sec. 2, popular Ethereum-style contracts often allow for a "logical split" of their state into disjointly owned components, which is much more fine-grained than assigning an entire contract to a single shard. This split makes it possible to process transactions affecting those contracts in parallel in different shards, thus providing a practical solution to scale up the network throughput.

Our contributions. The contributions of this work are:

- Identifying logical state ownership and operation commutativity as enabling mechanisms for sharding Ethereum-style contracts, and demonstrating adequacy of those notions for real-world Ethereum-style contracts (Sec. 2).
- A compositional static analysis that infers ownership and commutativity signatures for contracts written in Scilla [59] and translates them to shard allocation strategies (Sec. 3).
- An implementation of the analysis and of the algorithm for deriving sharding strategies in the tool called CoSplit and an end-to-end integration of CoSplit with a production-grade sharded blockchain protocol [48, 72] (Sec. 4).
- Evaluation of parallelism enabled by CoSplit-inferred signatures, demonstrating a consistent increase in system throughput with increasing the number of shards (Sec. 5).

2 Motivation and Key Ideas

2.1 Contract Usage in Ethereum

To motivate the design of our approach for sharding, we first present the trends for smart contract usage in Ethereum. Since there are over 700 million Ethereum transactions to date, processing all the execution traces is too computationally expensive. Therefore, we selected a random sample of 16,611 blocks, containing 1.1M transactions. This sample represents 0.17% of transactions and gives our measurements a 1% margin of error at a 99% confidence level. As the left plot in Fig. 1 shows, ordinary user-to-user transfers are on a solid downward trend. Moreover, single-contract transactions take up to 55% of the recent blocks in our sample.

In this work we focus on sharding single-contract transactions. The right plot in Fig. 1 shows the dominance of a specific type of such transactions that represent token transfers in a special kind of contract—ERC20 token contracts [26].

```
 1 contract ERC20 {
 2   mapping (address => uint256) balances;
 3   mapping (address => mapping (address => uint256)) allowances;
 4   /* Main public functions */
 5   function transfer(recipient, amount) {
 6     _transfer(_msgSender(), recipient, amount);
 7     return true;
 8   }
 9   function approve(spender, amount) {
10     _approve(_msgSender(), spender, amount);
11     return true;
12   }
13   function transferFrom(sender, recipient, amount) {
14     _transfer(sender, recipient, amount);
15     _approve(sender, _msgSender(),
16              allowances[sender][_msgSender()] - amount);
17     return true;
18   }
19   /* Auxiliary internal functions */
20   function _transfer(sender, recipient, amount) {
21     balances[sender] = balances[sender] - amount;
22     balances[recipient] = balances[recipient] + amount;
23   }
24   function _approve(owner, spender, amount) {
25     allowances[owner][spender] = amount;
26   } /* More functions */ }
```

Figure 2. A fragment of an ERC20 contract in Solidity.

ERC20 and other similar standardised contracts pose a big bottleneck to the network throughput: each of them requires *sequential processing* of all transactions that affect it.

2.2 Towards Sharding ERC20 Contract

Fig. 2 shows a fragment of the implementation of an ERC20 token contract [26] in Ethereum's high-level language Solidity [27]. The contract's replicated state is represented by two mutable fields: the mapping balances that contains data about the amount of tokens owned by token holders; and the mapping allowances that captures the amounts of tokens authorised for third-party transfers by their holders. The state manipulations are done by transactions initiated by users (*aka senders*) calling one of the functions: transfer for transferring tokens, approve for granting the transfer rights for a certain amount of tokens to a third party, and transferFrom for transferring tokens on behalf of the user identified as sender. The subtractions in lines 16 and 21 will fail if the approved spender (*resp.* the sender) does not have enough allowance, thus preventing double-spends.

The design of the ERC20 contract provides ample opportunities for shard-based parallelism. Consider the left part of Fig. 3 that shows a fragment of the *mutable* ERC20 state: the balances field mapping account addresses A–E (top part of each box) to the respective token balances (bottom part of the box), and allowances, which is a mapping from addresses (*e.g.*, A) to mappings of amounts allowed to transfer by third parties (*e.g.*, D and E) on their behalf. Now consider the following four single-contract transactions accessing that state concurrently by invoking functions from Fig. 2: tx_1 = $\mathsf{transfer}_A(B, v_1)$; tx_2 = $\mathsf{transfer}_C(A, v_2)$; tx_3 = $\mathsf{transferFrom}_D(A, C, v_3)$; tx_4 = $\mathsf{approve}_A(E, v_4)$. Here, a subscript denotes the transaction sender's address (accessed via _msgSender() in Fig. 2), while v_i stand for various

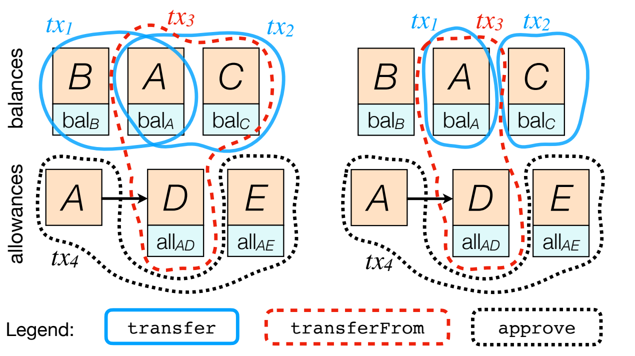


Figure 3. Two ways to identify footprints of four ERC20 transactions: via full affected state (left) and via state affected by non-commuting operations (right).

non-negative amounts, whose exact value is not important. All those transactions alter the contract state; the left part of Fig. 3 shows their corresponding *footprints*. It is easy to see that the footprints of, *e.g.*, tx_1 and tx_4 are *disjoint*, thus, their effects on the contract's state commute. Therefore, assuming the system provides an operation to *join* (*i.e.*, merge) updates on the "logically disjoint" state components, it should be possible to execute, *e.g.*, tx_1 and tx_4 in different shards.

Sharding Strategy 1: disjoint state ownership. Let us formulate the constraints for parallel execution of the transactions tx_1–tx_4 from Fig. 3, out of the knowledge that some of them commute, thanks to their footprint disjointness. We will denote by $\mathsf{Owns}(\mathcal{S}, \{\mathsf{f}_1, \ldots, \mathsf{f}_n\})$ an *ownership constraint*, meaning that the shard $\mathcal{S}$ logically owns the contract's state components (fields or map entries) $\mathsf{f}_1, \ldots, \mathsf{f}_n$ and, thus, *only this shard* may alter the values of those components by sequentially processing all the corresponding transactions.

Now consider two shards, $\mathcal{S}_1$ and $\mathcal{S}_2$ and the following set of ownership constraints, where bal and all denote the corresponding fields balance and allowances:

$$\left\{\begin{array}{l} \mathsf{Owns}(\mathcal{S}_1, \{\mathsf{bal}[A], \mathsf{bal}[B], \mathsf{bal}[C], \mathsf{all}[A][D]\}), \\ \mathsf{Owns}(\mathcal{S}_2, \{\mathsf{all}[A][E]\}) \end{array}\right\} \quad (1)$$

Clearly, $\mathcal{S}_1$ and $\mathcal{S}_2$ own disjoint portions of the contract's state, thus, it will be safe to assign transactions to shards as $\mathcal{S}_1 \mapsto \{tx_1, tx_2, tx_3\}$ and $\mathcal{S}_2 \mapsto \{tx_4\}$, obtain the final result deterministically by merging their non-conflicting changes. This sharding strategy scales for more shards with designated ownership of the contract's components and larger number of transactions with logically disjoint footprints.

Sharding Strategy 2: commutativity of addition. Even though transaction tx_2 modifies the component $\mathsf{bal}[A]$, it does so in a commutative fashion and, thus, cannot affect the outcome of any other of the listed transactions (ditto for tx_1 and $\mathsf{bal}[B]$). With this observation, we can refine the transaction footprints and the notion of ownership (Fig. 3, right), allowing for a parallel execution with three shards:

$$\left\{ \begin{array}{l} \mathsf{Owns}(\mathcal{S}_1, \{\mathsf{bal}[A], \mathsf{all}[A][D]\}), \\ \mathsf{Owns}(\mathcal{S}_2, \{\mathsf{all}[A][E]\}), \mathsf{Owns}(\mathcal{S}_3, \{\mathsf{bal}[C]\}) \end{array} \right\} \quad (2)$$

In the constraints above, $\mathcal{S}_1$ no longer owns $\mathsf{bal}[B]$ or $\mathsf{bal}[C]$, while shard $\mathcal{S}_3$ now owns $\mathsf{bal}[C]$. In order to perform transactions allocated as $\mathcal{S}_1 \mapsto \{tx_1, tx_3\}$, $\mathcal{S}_2 \mapsto \{tx_4\}$, $\mathcal{S}_3 \mapsto \{tx_2\}$, obtaining the *same result* as in the previous case, we need to redefine the state *join* operation. Specifically, instead of *overwriting* the values in entries $\mathsf{bal}[B]$ and $\mathsf{bal}[A]$ upon "disjoint merging" as before, we will need to *add up the deltas* to those components resulting from token transfers in tx_1 and tx_3, similarly to handling ordinary transfers (Sec. 1).

The main idea. To summarise these observations: contracts such as ERC20, whose operations only manipulate a *small part* of the state, allow for parallel conflict-free execution of their operations, if these operations commute. The *ownership constraints* state *which parts* of a contract's state a shard must have exclusive access to in order to execute its operations without conflicts with other shards altering the same contract concurrently. The *join* defines the way to deterministically reconcile outcomes of the parallel executions.

2.3 Commutativity and State Ownership

It is common to reason about operation commutativity in terms of action traces [14]. That said, our way of thinking is inspired by the logical abstractions used for compositional verification of heap-manipulating programs [10, 11, 53].

In our setup, we are interested in parallelising executions of a family of single-contract transactions over a state-space Σ of a contract, collectively represented by a function $\mathcal{F}_x : \Sigma \longrightarrow \Sigma$. Here, x denotes a vector of user inputs, *i.e.*, specifying which contract's function to call, as well as its inputs. Two transactions identified by different user inputs x_1 and x_2 commute *iff* for any state σ, $\mathcal{F}_{x_1}(\mathcal{F}_{x_2}(\sigma)) = \mathcal{F}_{x_2}(\mathcal{F}_{x_1}(\sigma))$.

In order to enable parallelism, our goal is to identify a *commutative*, *associative*, and *partial* operation $\uplus : \Sigma \longrightarrow \Sigma \longrightarrow \Sigma$, such that for any σ_1, σ_2 and x, if $\mathcal{F}_x(\sigma_1)$ is defined (*i.e.*, σ_1 contains at least the *footprint* of $\mathcal{F}_x$) and $\sigma_1 \uplus \sigma_2$ is defined, then $\mathcal{F}_x(\sigma_1 \uplus \sigma_2) = \mathcal{F}_x(\sigma_1) \uplus \sigma_2$. This equality is referred to as *action locality* in the program logics literature [11], and, when it holds, it enables compositional program analyses [23] and concurrency specifications [47].[1] The virtue of $\mathcal{F}_x$'s locality for our purposes becomes apparent by observing the following chain of equalities for $\sigma = \sigma_1 \uplus \sigma_2$ when σ_1 and σ_2 are such that $\mathcal{F}_{x_1}(\sigma_1)$ and $\mathcal{F}_{x_2}(\sigma_2)$ are defined:

$$\begin{aligned} \mathcal{F}_{x_1}(\mathcal{F}_{x_2}(\sigma)) &= \mathcal{F}_{x_1}(\mathcal{F}_{x_2}(\sigma_1 \uplus \sigma_2)) = \mathcal{F}_{x_1}(\sigma_1 \uplus \mathcal{F}_{x_2}(\sigma_2)) \\ &= \mathcal{F}_{x_2}(\sigma_2) \uplus \mathcal{F}_{x_1}(\sigma_1) = \mathcal{F}_{x_2}(\sigma_2 \uplus \mathcal{F}_{x_1}(\sigma_1)) \\ &= \mathcal{F}_{x_2}(\mathcal{F}_{x_1}(\sigma_2 \uplus \sigma_1)) = \mathcal{F}_{x_2}(\mathcal{F}_{x_1}(\sigma)) \end{aligned}$$

[1] Readers familiar with state-of-the art program logics for concurrency can recognise that we are looking for a suitable Partial Commutative Monoid (PCM) [11, 37, 47], which would enable *framing* of contract operations $\mathcal{F}_x$.

This reasoning demonstrates the desired commutativity, and also provides a recipe for computing the final result in a *divide-and-conquer* fashion by taking it to be $\mathcal{F}_{x_1}(\sigma_1) \uplus \mathcal{F}_{x_2}(\sigma_2)$; the order does not matter, as $\uplus$ is commutative and associative. One can think of $\uplus$ as both the "logical split and join" operations, while a footprint of a transaction executing $\mathcal{F}_x$ is the minimal part σ' of the contract state, which *must be owned* by the shard executing it, so that $\mathcal{F}_x(\sigma')$ is defined.

Getting back to our motivating example of ERC20 sharding, **Strategy 1** corresponds to $\uplus$ taken as a disjoint union of the entry sets of the contract's mapping fields (let's call it OwnOverwrite). **Strategy 2** corresponds to $\uplus$ defined as a non-disjoint union with an implicit split of integer values in map entries—this way in the case of concurrent updates, the result can be obtained by summation of the per-shard portions of those values (we will call it IntMerge).

2.4 Pragmatic Considerations and Technical Setup

In the PL community the notion of "smart contracts" is almost synonymous with Ethereum, due to its prevalence in this area. While it would be desirable to implement the above ideas directly in Ethereum, unfortunately, its infrastructure is currently unsuitable for our purposes:

- **Protocol-level support.** At the time of writing, the available prototype of Ethereum 2.0 [68] does not support cross-shard transactions or smart contracts.
- **Language-level support.** EVM bytecode, Ethereum's low-level language, is difficult to analyse soundly, due to the lack of modularity and structured control flow [69]. One could engineer an analysis inferring ownership constraints for Solidity [27], a high-level Ethereum language, by possibly decompiling EVM contracts [31]. However, a number of Solidity's features (*e.g.*, inter-contract calls) and unpredicateble performance of EVM decompilers make it a challenging target for an efficient sound static analysis [35].

Bringing Ethereum infrastructure to the state necessary to deploy the described ideas would be an effort going well beyond the scope of this paper. Instead, we implemented our approach as a static analysis for Scilla [59]—a strongly-typed ML-style language for smart contracts. Scilla is supported natively (via a definitional interpreter) by an industry-scale blockchain [48, 72] that (a) provides rudimentary infrastructure for sharding, (b) is available open-source, and (c) is widely used and contains dozens of contracts implemented in Scilla by users and available for evaluating our approach.

3 CoSplit Analysis in a Nutshell

3.1 The Language

Scilla [59] is a minimalistic memory- and type-safe functional language, similar to OCaml and Haskell for an account-based (*i.e.*, Ethereum-style) smart contract model. Scilla provides a very small set of state-manipulating primitives for altering contract state (*i.e.*, reading from the blockchain state and changing the values of contract fields). Its *pure* (*i.e.*,

$$
\begin{array}{llcl}
\text{(identifiers)} & i, f, c & ::= & \text{alpha-numeric string} \\
\text{(types)} & t & ::= & \mathit{int} \mid \mathsf{string} \mid \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{map}\ t\ t \mid t \to t \mid \ldots \\
\text{(patterns)} & \mathit{pat} & ::= & \_ \mid i \mid \mathbf{constr}\ c\ \overline{\mathit{pat}} \\
\text{(expressions)} & e & ::= & \mathbf{val}\ v \mid \mathbf{var}\ i \mid \mathbf{message}\ (\bar{i} \mapsto \bar{i}) \mid \mathbf{constr}\ c\ \bar{t}\ \bar{i} \mid \\
 & & & \mathbf{builtin}\ \mathit{blt}\ \bar{i} \mid \mathbf{let}\ i = e_1\ \mathbf{in}\ e_2 \mid \mathbf{fun}\ (i : t) \Rightarrow e \mid \\
 & & & \mathbf{app}\ i\ \overline{i_j} \mid \mathbf{match}\ i\ \overline{\mathit{pat} \Rightarrow \bar{e}} \mid \mathbf{tfun}\ \alpha \Rightarrow e \mid \mathbf{inst}\ i\ \bar{t} \\
\text{(statements)} & s & ::= & i_1 \leftarrow i_2 \mid i_1 := i_2 \mid i = e \mid i_1\ \overline{[i_k]}\ := i_2 \mid i_1 \leftarrow i_2\ \overline{[i_k]} \mid \\
 & & & i_1 \leftarrow \mathbf{exists}\ i_2\ \overline{[i_k]} \mid \mathbf{delete}\ i_1\ \overline{[i_k]} \mid i_1 \leftarrow \&i_2 \mid \\
 & & & \mathbf{match}\ i\ \overline{\mathit{pat} \Rightarrow \bar{s}} \mid \mathbf{accept} \mid \mathbf{send}\ i \mid \mathbf{event}\ i \mid \mathbf{throw}
\end{array}
$$

Figure 4. Scilla syntax.

side effect-free) fragment corresponds to System F [29, 57] without recursion (but with bounded iteration). All of the standard library as well as user-defined contract-agnostic computations are implemented in Scilla as pure functions. This design choice removes the need for inter-contract calls for the sake of code reuse and makes contract analysis scale, as pure functions need to be analysed *only once*.

Contracts in Scilla are encoded as communicating state-transition systems in the style of IO-automata [49]. That is, all interaction between contracts is done by means of *message passing*. A contract's state changes as a result of executing its ***transitions*** as reactions to received messages from the users or other contracts. This model allows one to analyse each contract's transitions *in isolation* from any other contract's code, thus allowing for deriving their signatures *statically* without over-approximating the effects of the external calls.

Fig. 4 describes the syntax for Scilla statements and expressions, and Fig. 5 shows the code for the `Transfer` transition of a FungibleToken (*i.e.*, ERC20) contract. The transition takes one implicit argument, the address `_sender` of the transaction originator, and two explicit arguments, the address of the recipient and the amount of tokens to be transferred. It reads the balance of the sender (line 2), ensures that the sender has enough tokens to transfer (line 6), and updates the sender (line 9) and recipient balances (line 14).

3.2 Inferring Transition Summaries

CoSplit implements a compositional inter-procedural *effect analysis* [52] inferring *transition summaries* (a set of *effects*), which describe, in a symbolic form, (1) the *state footprint* of transitions and (2) the *contributions* of the initial values of contract fields, read during a transition's execution, to the final field values. A transition's summary is the result of the abstract interpretation [16, 17] of the transition's code. These summaries help answer queries such as (1) "Does the transition write to field f?" and (2) "Can the transition's effect to field f be represented as an addition of a constant to f's old value?". The two types of queries correspond to our two sharding strategies (see the discussion in Sec. 2.2).

The *state footprint* describes the portion of a contract state affected by a transition (and, respectively, by a transaction that executes it), whereas the *contribution types* let us decide if writes that affect the same piece of state commute. We explain the analysis in two stages, corresponding to the two

```
1  transition Transfer(to: ByStr20, amount: Uint)
2    get_from_bal ← balances[_sender];
3    match get_from_bal with
4    | Some bal ⇒
5      can_do = uint_le amount bal;
6      match can_do with
7      | True ⇒
8        new_from_bal = builtin sub bal amount;
9        balances[_sender] := new_from_bal;
10       get_to_bal ← balances[to];
11       new_to_bal = (match get_to_bal with
12       | Some bal ⇒ builtin add bal amount
13       | None ⇒ amount);
14       balances[to] := new_to_bal
15     | False ⇒ (* Report a transaction failure *)
16   msg_to_recipient = {_tag : "Accepted"; _to : to; ...};
17   msg_to_sender = {_tag : "Success"; _to : _sender; ...};
18   send msg_to_recipient; send msg_to_sender
```

Figure 5. FungibleToken `Transfer` transition in Scilla.

types of queries. The stages are intertwined in the implementation, but here we separate them to aid in presentation.

3.3 State Footprints

Fig. 6 shows the CoSplit abstract domain. The first stage of the analysis computes an over-approximation of the state footprints of contract transitions, expressed as a set of *effects* (denoted ε). Effects describe how the transition interacts with the blockchain state. For instance, the AcceptFunds effect (contributed by the **accept** statement) changes *both* the contract's and the sender's native token balances. Similarly, the SendMsg effect (contributed by **send**) might invoke transitions of other contracts or send native tokens. Finally, the Read and Write effects describe which portions of the contract's own state may be accessed by the transition. For each transition, CoSplit iterates over every statement in the transition's code and determines the static over-approximation of that statement's effect. In some cases, this over-approximation is the uninformative effect $\top$. Due to Scilla's design, the translation between statements and effects is direct. As an example, we show the analysis rules for map reads and writes, which can be found in the top box of Fig. 7. These rules are applied when analysing lines 2, 9, and 14 in Fig. 5. The parts shown in grey boxes, including the rule for the non-effectul Bind statement, appear due to the second stage of the analysis, explained below.

The MapGet and MapUpdate rules extend the transition's summary Σ with the appropriate effect ε whenever the keys $\overline{i_k}$ used to index into the map are *transition parameters*, as they are in Fig. 5. If the access cannot be summarised, the $\top$ effect is given. Generally, an access can be statically described when the keys are not dependent on the contract state, but we limit ourselves to keys that are transition parameters to simplify transaction dispatch, which is described in Sec. 4.3.

3.4 Contribution Types

The summaries produced so far are sufficient to enable the disjoint state ownership sharding strategy. But, as we have seen, we can execute in parallel even transactions with *effects*

$$
\begin{array}{llcl}
\text{(effect)} & \varepsilon & ::= & \mathsf{Read}(\mathsf{f}) \mid \mathsf{Write}(\mathsf{f}, \tau) \mid \mathsf{Condition}(\tau) \mid \\
 & & & \mathsf{AcceptFunds} \mid \mathsf{SendMsg}(\tau) \mid \top \\
\text{(contrib. type)} & \tau & ::= & \langle \overline{\mathrm{cs} \mapsto (\mathrm{card}, \mathrm{ops})}, p \rangle \mid \mathbf{EFun}\ i\ \tau \mid \top \mid \bot \\
\text{(contrib. src.)} & \mathrm{cs} & ::= & \mathbf{Field}\ \mathsf{f} \mid \mathbf{Const}\ c \mid \mathbf{Formal}\ i \mid \top \\
\text{(cardinality)} & \mathrm{card} & ::= & 0 \mid 1 \mid \omega \\
\text{(precision)} & p & ::= & \mathrm{Exact} \mid \mathrm{Inexact} \\
\text{(operation)} & \mathrm{op} & ::= & \mathbf{Builtin}\ blt \mid \mathbf{Cond}
\end{array}
$$

$$
\begin{array}{lll}
\text{(cardinality order)} & 0 \sqsubseteq 1 \sqsubseteq \omega & \\
\text{(operations order)} & \mathrm{ops}_1 \sqsubseteq \mathrm{ops}_2 & \text{iff } \mathrm{ops}_1 \subset \mathrm{ops}_2 \\
\text{(precision order)} & \mathrm{Exact} \sqsubseteq \mathrm{Inexact} &
\end{array}
$$

$$
\begin{array}{lll}
0 \oplus \alpha = \alpha & 0 \sqcup \alpha = \alpha & 0 \otimes \alpha = 0 \\
1 \oplus 1 = \omega & 1 \sqcup 1 = 1 & 1 \otimes 1 = 1 \\
\alpha \oplus \omega = \omega & \alpha \sqcup \omega = \omega & \alpha \otimes \omega = \omega
\end{array}
$$

Figure 6. Components of CoSplit abstract domain.

over the same state, as long as the effects commute. The second stage of the analysis annotates the effects produced by the first stage with contribution types (denoted τ), which help determining whether they commute. Concretely, for every expression e, CoSplit computes a *contribution type*, an over-approximation of the set of arithmetic operations and of the set of contract state components from the beginning of the transition execution that contribute to e's result.

The type τ (*cf.* Fig. 6) ascribed to an expression e indicates, which *contribution sources*, *i.e.*, parts of the contract state (**Field** f), transition parameters/constants (**Const** c), and function parameters (**Formal** i), flow into e's result, what operations are applied to those sources, and *how many times* each source contributes to e's result. The precision component p in types records whether over-approximation of the set of operations has taken place due to joining control flows, *i.e.*, the analysis has lost precision. This lets us answer questions like "Can the transition's effect to field f be represented as an *addition of a constant* to f's old value?". If the τ ascribed to the value written to field f has as its only *exact* contribution **Field** f $\mapsto$ (1, **Builtin** *add*), and also some constants and transition parameters, the answer is *yes*.

The importance of cardinalities. The most important component in the type is the *cardinality* of **Field** f. If f did not show up in τ, then the written value would be a constant, and different writes would not commute, as they might have potentially different values (*e.g.*, different transition parameters). Conversely, if f's contribution was non-linear, *i.e.*, its cardinality is ω ("many"),[2] then even though f is modified by a commutative operation (addition), the effect would not commute. For example, the linear function $f(x) = x+1$ commutes with $g(x) = x+2$, but does not with $h(x) = x+x+1$, as $f(h(a)) \neq h(f(a))$. The *linearity* ("used-once") information attached to contribution sources lets us ensure that operations are used in ways that guarantee commutative effects. We lift the $\oplus$ operation on cardinalities to types by adding

[2]Our cardinality domain is inspired by GHC's cardinality analysis [60].

$$
\textsc{Bind}\ \frac{\Gamma \vdash e : \tau \quad \Gamma' = \Gamma, i : \tau}{\Gamma; \Sigma \vdash i = e \rightsquigarrow \Gamma'; \Sigma}
\qquad
\textsc{MapUpdate}\ \frac{\Gamma \vdash i_2 : \tau \quad b = \mathrm{CanSummarise}\ \overline{i_k} \\ \varepsilon = \mathbf{if}\ b\ \mathbf{then}\ \mathsf{Write}(i_1\ \overline{[i_k]}, \tau)\ \mathbf{else}\ \top}{\Gamma; \Sigma \vdash i_1\ \overline{[i_k]} := i_2 \rightsquigarrow \Gamma; \Sigma, \varepsilon}
$$

$$
\textsc{MapGet}\ \frac{b = \left(\mathsf{Write}(i_2\ \overline{[i_k]}, \_) \notin \Sigma\right) \wedge \left(\mathrm{CanSummarise}\ \overline{i_k}\right) \\ \tau, \varepsilon = \mathbf{if}\ b\ \mathbf{then}\ \langle \mathbf{Field}\ i_2\ \overline{[i_k]} \mapsto (1, \varnothing), \mathrm{Exact} \rangle, \mathsf{Read}(i_2\ \overline{[i_k]})\ \mathbf{else}\ \top, \top}{\Gamma; \Sigma \vdash i_1 \leftarrow i_2\ \overline{[i_k]} \rightsquigarrow \Gamma, i_1 : \tau; \Sigma, \varepsilon}
$$

$$
\textsc{Literal}\ \Gamma \vdash \mathbf{val}\ l : \langle \mathbf{Const}\ l \mapsto (1, \varnothing), \mathrm{Exact} \rangle
\qquad
\textsc{Constr}\ \frac{\tau = \bigoplus \overline{\Gamma(i)}}{\Gamma \vdash \mathbf{constr}\ c\ \overline{t}\ \overline{i} : \tau}
$$

$$
\textsc{Builtin}\ \frac{\tau' = \bigoplus \overline{\Gamma(arg)} \\ \tau = \tau' \text{ with ops } \mathrel{+}= blt}{\Gamma \vdash \mathbf{builtin}\ blt\ \overline{arg} : \tau}
\qquad
\textsc{Fun}\ \frac{i = \text{unique arg. identifier} \\ \Gamma' = \Gamma, i : \langle \mathbf{Formal}\ i \mapsto (1, \varnothing), \mathrm{Exact} \rangle \\ \Gamma' \vdash e : \tau_e \quad \tau = \mathbf{EFun}\ i\ \tau_e}{\Gamma \vdash \mathbf{fun}\ (i : t) \Rightarrow e : \tau}
$$

$$
\textsc{Let}\ \frac{\Gamma' = \Gamma, i : \tau_1 \\ \Gamma \vdash e_1 : \tau_1 \quad \Gamma' \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ i = e_1\ \mathbf{in}\ e_2 : \tau}
\qquad
\textsc{App}\ \frac{i_j = \text{unique identifier for } j\text{-th arg.} \\ \Gamma \vdash f : \tau_f \quad \forall j, \Gamma \vdash a_j : \tau_j \\ \tau = \bigoplus (\tau_f[i_j] \otimes \tau_j)}{\Gamma \vdash \mathbf{app}\ f\ \overline{a_j} : \tau}
$$

$$
\textsc{Match}\ \frac{\Gamma \vdash x : \tau_x \quad \forall i, \Gamma_i = \overline{\mathrm{binder}(pat_i) \to \Gamma(x)} \\ \forall i, \Gamma \cup \Gamma_i \vdash e_i : \tau_i \quad \tau = \mathrm{MatchC}(x, \tau_x, pat_i, e_i, \overline{\tau_i})}{\Gamma \vdash \mathbf{match}\ x\ \overline{pat_i} \Rightarrow \overline{e_i} : \tau}
$$

Figure 7. Selected rules for statements and expressions.

the cardinalities of matching sources and set-unioning their operations, ascribing to the result the $\sqcup$ of their precisions. For $\otimes$, which is defined only between a type and a single contribution, we multiply the cardinalities of the arguments and modify the other components analogously to the $\oplus$ lifting.

Computing contribution types. The values read from the mutable contract state, literals, transition and contract parameters are all contribution sources. For example, the Literal and MapGet rules in Fig. 7 show how new contribution sources are introduced. A read into a binder i_1 from a location that was not overwritten, extends the typing context Γ by giving i_1 the linear contribution type τ shown in the rule. The type shows that the value of i_1 is the value of the respective "pseudo-field" $i_2\ \overline{[i_k]}$ (*i.e.*, a map entry) at the beginning of the transition execution. Contributions from multiple sources are combined with their cardinalities added up *point-wise* via $\oplus$ operator (*cf.* Fig. 6). For example, the application of a Builtin function adds up the contributions from its arguments and records an application of **Builtin** *blt* to each contribution source, recording this in the type.

Functions are given *arrow types*, **EFun** $i\ \tau$, with each formal parameter constituting an artificial contribution source **Formal** i. When the function is applied (rule App), the arrow type is evaluated by substituting the formal parameter's contribution with the actual argument's contributions, multiplying the cardinalities. Our approach supports up to

$$
\begin{array}{ll}
\text{Read}(\text{balances}[\_\text{sender}]) & \text{Read}(\text{balances}[\text{to}]) \\
\text{Condition}(\text{balances}[\_\text{sender}],\ \text{amount}) & \\
\text{Write}(\text{balances}[\_\text{sender}],\ \langle \text{amount\&balances}[\_\text{sender}],\ 1,\ \text{sub}\rangle) & \\
\text{Write}(\text{balances}[\text{to}],\ \langle \text{amount\&balances}[\text{to}],\ 1,\ \text{add}\rangle) & \\
\text{SendMsg}(\text{zero}; \text{to}) & \text{SendMsg}(\text{zero}; \_\text{sender})
\end{array}
$$

Figure 8. Set of effects of the `Transfer` transition.

$$
\begin{array}{lll}
\text{(constraint)} & oc ::= & \text{Owns}(\text{f}) \mid \text{UserAddr}(x) \mid \text{NoAliases}(\langle x, y\rangle) \mid \\
 & & \text{SenderShard} \mid \text{ContractShard} \mid \bot \\
\text{(join)} & \uplus_{\text{f}} ::= & \text{OwnOverwrite} \mid \text{IntMerge}
\end{array}
$$

Effect	Constraint
$\top \vee \text{SendMsg}(\top)$	$\bot$
AcceptFunds	SenderShard
SendMsg(_amount.$\tau \neq 0$)	ContractShard
SendMsg(_recipient.$\tau = r$)	UserAddr(r)
Read/Write(m [x]), Read/Write(m [y])	NoAliases($\langle x, y\rangle$)

Figure 9. Elements of sharding signatures (top). Translation between effects and non-ownership constraints (bottom).

second-order Scilla functions by expressing operations on contribution types ($\oplus$, $\sqcup$, and $\otimes$) at the type level, and deferring normalisation until the arguments are known. For simplicity, we do not show this in the figures. Even though Scilla provides polymorphic functions and type applications (as in System F), they bear no significance for our analysis.

Finally, the types given to Match over-approximate the contributions of all clauses, whose types are combined using the MatchC operator defined as follows:

$$
\begin{array}{rcl}
\text{MatchC}(x, \tau_x, \overline{pat_i}, \overline{e_i}, \overline{\tau_i}) & \triangleq & \tau_{cond} \oplus \bigsqcup \overline{\tau_i} \\
 & \text{where} & \\
\tau_{cond} & \triangleq & \textbf{if } \text{IsKnownOp}(x, \overline{pat_i}, \overline{e_i}) \\
 & & \textbf{then } \bot \textbf{ else } \text{AdaptC } \tau_x \\
\text{AdaptC } \langle \overline{\text{cs} \mapsto (\_, \_)}, \_\rangle & \triangleq & \textbf{if } \text{SameVars}(\overline{\tau_i}) \\
 & & \textbf{then } \langle \overline{\text{cs} \mapsto (0, \textbf{Cond})}, \text{Exact}\rangle \\
 & & \textbf{else } \langle \overline{\text{cs} \mapsto (0, \textbf{Cond})}, \text{Inexact}\rangle
\end{array}
$$

The additional contribution τ_{cond} accounts for whether **match**ing over the scrutinee x induces non-trivial data flow (in which case its contribution is determined via AdaptC), or simply "peels off" a constructor of an `option` value (in which case it has no contribution). The latter is a very common special scenario, (see, *e.g.*, lines 11–13 of Fig. 5), and without this machinery the analysis would lose too much precision.

3.5 Calculating Sharding Signatures

From the set of transition summaries of a given contract (one such summary is shown in Fig. 8), and provided with user input as to what transitions to attempt to shard and which fields can be treated *weakly* for reading (*cf.* Sec. 4.2.3), CoSplit derives a sharding signature, consisting of a set of constraints $\overline{oc}$ for each transition in the contract and a join operation $\uplus_{\text{f}}$ for each field. The top part of Fig. 9 enumerates the constraints that summaries can impose, as well as join operations we currently support. Constraints are static symbolic representations of conditions that *must be satisfied at runtime*. They refer to mutable fields or transition parameters as symbolic values, *e.g.*, Owns(f) and UserAddr(x). For instance, the Owns(balances[_sender]) constraint denotes that a shard executing the transition must own the _sender portion of the balances state component, where _sender is replaced at runtime by the actual value given by the transaction. The other constraints are imposed by the blockchain environment, *e.g.*, SenderShard, which must be satisfied if the contract accepts funds, or arise as preconditions for the soundness of our analysis, *e.g.*, that keys used for map accesses do not alias. $\bot$ corresponds to an unsatisfiable constraint, meaning that the transition cannot be executed in parallel with other transactions over the same contract.

Algorithm 3.1: Derive Sharding Signature

```
input  : effect summaries, selected transitions, weak reads
output : transition (ownership) constraints, field join operations
$\overline{\Sigma} \leftarrow$ effect summaries of selected transitions
wrs $\leftarrow$ reads the user accepted might be stale
for $\Sigma \in \overline{\Sigma}$ do
    cfs $\leftarrow$ GetConstantFields($\Sigma$)
    $\Sigma \leftarrow \forall f \in$ cfs , $\Sigma$.remove(Read($f$))
    $\Sigma \leftarrow \Sigma$.MarkConstantsInTypes(cfs)
lcws $\leftarrow$ GetTransitionCommWrites($\overline{\Sigma}$)
cws, joins $\leftarrow$ TryConsolidateJoinsGlobally(lcws)
$\overline{\Sigma} \leftarrow \overline{\Sigma}$.RemoveSpuriousReads(cws)
if joins $\neq \varnothing \wedge$ wrs = StaleReads($\overline{\Sigma}$, joins) then
    oc $\leftarrow$ {}
    for $\Sigma \in \overline{\Sigma}$ do
        c $\leftarrow$ GenEnvironmentConstraints($\Sigma$)
        foreach Read(f) $\in \Sigma$ do c $\cup$ Owns(f)
        foreach Write(f) $\in (\Sigma \setminus$ cws) do c $\cup$ Owns(f)
        oc $\leftarrow$ oc $\cup$ c
    return (oc, joins)
```

Algorithm 3.1 shows the procedure for deriving the sharding signature. The contract developer selects a set of transitions that should be executed in parallel, and the algorithm inspects their summaries to determine what constraints must be satisfied to enable parallel execution. First, it identifies which (if any) contract fields are not written to in the selected transitions, and marks their reads as non-effectful (*i.e.*, it removes them from the summary) and their contributions as constant. Second, looking at each summary in isolation, it identifies using the types, which writes have a commutative effect, *e.g.* Write(balances[to]) in Fig. 8. Then, it determines if a join operation exists for every field, *i.e.* the writes across transitions are compatible, and marks reads that only flow into commutative writes, *e.g.* Read(balances[to]) (since `balances[to]` does not appear in any other type), as non-effectul.[3] Finally, if the developer accepts that reads from fields that are commutatively written to might return *stale data* (*cf.* Sec. 4.2.3), the algorithm translates effects into constraints via the mapping in Fig. 9 and by requiring ownership of every field that is read or non-commutatively written to.

[3]The Condition effect prevents removal of reads that affect control-flow.

4 Enabling Parallelism with CoSplit

In this section, we show how the signatures inferred by CoSplit, as described in Sec. 3, can be used to allow for parallel transaction execution in a sharded blockchain.

4.1 The Sharding Model

We integrated CoSplit with Zilliqa blockchain [72]. Zilliqa is one of the first sharded chains in production. It implements the Elastico protocol for secure sharding [48] and relies on an optimised version of the Practical Byzantine Fault Tolerance (PBFT) protocol for consensus in the network [13, 64]. At the time of writing, Zilliqa mainnet has processed 9.6 million transactions and contained 28 types of *unique* smart contracts (some of them have many deployed copies). Below, we outline the relevant parts of its architecture and transaction processing logic, referring the reader to the corresponding manuscripts for details (on, *e.g.*, security and epoch-based mining) [48, 72], which are not critical for our presentation.

Network architecture. The Zilliqa network consists of three main components: the *lookup nodes*, the *shards*, and the *Directory Service committee* (*aka* the DS committee) (*cf.* Fig. 10).

Lookup nodes are the entry-point to the network. Any transaction created by a user has to be sent to the lookup nodes, which thereupon group several transactions together in a packet and dispatch them to one of the shards or the DS committee for processing. Each shard (and similarly the DS committee, which in fact is a special shard) runs PBFT to reach consensus on validated transactions. It then proposes a MicroBlock (MB) that contains information on the transactions that it has processed. MicroBlocks are then sent to the DS committee together with StateDelta (SD) which encodes the changes in the state of the accounts that were touched by the transactions within a MicroBlock. Once all the MicroBlocks and the corresponding StateDeltas reach the DS committee, the latter combines them all in the form of a FinalBlock (FB) and a FinalStateDelta (FSD). The FinalBlock and FinalBlockDelta are then sent back to each shard so that all the shards have the same view of the global state.

The default sharding strategy. A client-issued transaction can be processed either by one of the shards or by the DS committee (Fig. 10). Zilliqa employs a simple deterministic transaction assignment strategy to shards to ensure that double spends are detected within a shard without complex cross-shard communication [42]. User-to-user payment transactions are deterministically assigned to shards based on the sender's address. That is, all transactions from the same user get handled in the same shard, so any double spend from a specific user can be detected within a single shard in the same way it gets handled in a non-sharded architecture.

For smart contracts, Zilliqa implements a non-efficient *conservative* strategy. Specifically, the network statically assigns both contracts and end users to shards. Transactions to a contract invoked by users residing in the same shard as the contract are handled within the shard, while transactions to a contract invoked by users from an outside shard are handled in the DS committee. To ensure that shards and the DS do not end up manipulating the state of the same contract concurrently, the protocol requires the DS committee to process transactions assigned to it *only after* the shards have finished processing their transactions.

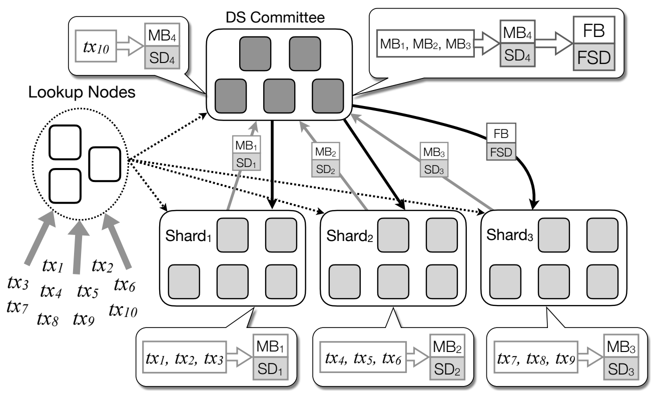


Figure 10. Transaction processing in a sharded architecture.

Given this simple deterministic assignment, the parallelism achieved for smart contract transaction processing is quite limited. In fact, the more shards there are, the more transactions will need to be processed by the DS committee.

4.2 Revising the Account-Based Blockchain Model

In order to employ the described sharding model for CoSplit-enabled parallelism, we need to revise a few core aspects in the design of Ethereum-style blockchains.

4.2.1 Relaxing the nonce mechanism. Ethereum's account based model [63] (adopted by Zilliqa and similar systems) uses of the *nonce* mechanism for defining a total order on all transactions emitted by a particular user [55]. Nonces are calculated by counting the number of transactions sent from a user address and are digitally signed, addressing the following design aspects: (a) strict, gap-free, user-defined ordering of transactions, and (b) prevention of replay attacks. Thanks to the nonces, the user can send many transactions with subsequent numbers, and they are going to be processed in *this exact order*—the protocol will not process transaction with a nonce $n + 1$ before the one with nonce n.

Because of aspect (a), nonces pose a bottleneck to sharded executions. While in plain Zilliqa all transactions from a single user are guaranteed to be handled in the same shard, the nonce mechanism prevents parallel executing of transactions with the same origin in different shards, as the total order of nonces cannot be communicated. We notice that in practice no applications rely on a specific order of user transactions *before they are committed* by the protocol.[4] Therefore, it suffices for transactions to be processed in an increasing nonce

[4]Furthermore, the UTxO blockchain model adopted by, *e.g.*, Bitcoin [51] promotes this kind of weak notion of consistency, in which the user *cannot* predict the order in which her transactions are committed.

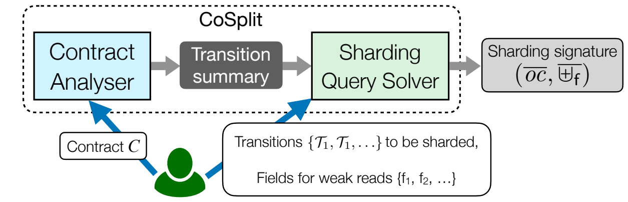

CoSplit
Contract Analyser
Transition summary
Sharding Query Solver
Sharding signature $(\overline{oc}, \overline{\uplus}_\mathsf{f})$
Contract C
Transitions $\{\mathcal{T}_1, \mathcal{T}_1, \ldots\}$ to be sharded,
Fields for weak reads $\{\mathsf{f}_1, \mathsf{f}_2, \ldots\}$

Figure 11. CoSplit for a contract developer (offline mode).

order, without waiting for all "gaps" to be filled, treating them similarly to ballots in Paxos [44]. This relaxation requires a very small change in the protocol logic. With it, we kept the aspect (b) of the nonce mechanism, while allowing for parallel executions. For instance, this way we can execute in parallel two disjoint sets of *commuting* transactions from the same user with nonces $\{1, 3, 5\}$ and $\{2, 4\}$, respectively.

4.2.2 Parallel gas accounting. *Gas accounting* is a mechanism to charge users for executing transactions [69]. Such deductions must be treated sequentially to avoid overspending. We circumvent this bottleneck to parallelism by splitting a user's balance across shards (with a larger fraction given to the shard handling money transfers from that user), so gas costs can be charged without coordinating balance changes.

4.2.3 Weak reads. In the `Transfer` transition (Fig. 8), we saw that the write into `balances[to]` has a commutative effect. As a result, the processing shard does not need to own the field to execute the transition. However, allowing commutative writes means that transitions executing in a different shard might read *stale* values of `balances[_sender]`. In our example, this is fine—the sender may have more tokens that she thinks. Yet, in general, introducing commutative writes weakens the semantics of reads and a static analysis cannot determine whether this is "fine" for specific contracts. For this, we need user input. Ideally, the programming language itself would allow one to mark certain reads as *weak*, but neither Scilla nor Solidity, both designed with sequential semantics in mind, currently have this feature. For now, weak reads must be provided as input to Algorithm 3.1.

4.3 CoSplit in Action

Using CoSplit assumes two modes: offline and online. In the former, the user who is about to deploy her contract to the blockchain, provides hints to the tool in order to choose the most suitable sharding signature (Fig. 11). In the latter, CoSplit is run automatically by the miners as a part of the validation pipeline for contracts proposed to be deployed.

Offline mode. Prior to the deployment of her contract C (written in Scilla), the developer runs the CoSplit analysis on C's code, producing a *transition summary*. Such a summary (*cf.* Sec. 3.2) itself is not yet useful for sharding. The developer next sends a query (a (sub)set $\{\mathcal{T}_1, \mathcal{T}_2, \ldots\}$ of C's transitions and a subset $\{\mathsf{f}_1, \mathsf{f}_2, \ldots\}$ of C's fields that can be treated *weakly* for reading) to the sharding solver, obtaining as a result a sharding signature: a set of ownership constraints $\overline{oc}$ and a dictionary of *per-field* join functions $\overline{\uplus}_\mathsf{f}$ suitable for sharding those transitions. CoSplit returns a pair $(\overline{oc}, \overline{\uplus}_\mathsf{f})$ of encodings, which the developer submits along with C's code in a contract-deploying transaction.

Validating sharding signatures. Each node in the shard, upon receiving a transaction with a contract C and its signature $(\overline{oc}, \overline{\uplus}_\mathsf{f})$, runs CoSplit ensuring that the signature is valid (the set of sharding transitions can be obtained from the constraints $\overline{oc}$). The signature is then broadcast, along with the contract code and metadata, to all nodes in the system.

Assigning transactions to shards. A lookup node, upon receiving a transaction $tx = \langle C, \mathcal{T}, x\rangle$ invoking a transition $\mathcal{T}$ of a contract C with inputs x, first conservatively checks whether the transaction is indeed single-contract (*i.e.*, no more than one transition is invoked). If it cannot validate it as such, the transaction is routed to the DS committee. Otherwise, the lookup node exercises the *sharding strategy* induced by C's signature $(\overline{oc}, \overline{\uplus}_\mathsf{f})$, if such signature is available.

To do so, the lookup node invokes the $\mathit{dispatch}_{\overline{oc}}(\mathcal{T}, x)$ procedure, which identifies a shard $\mathcal{S}$ that satisfies $\mathcal{T}$'s constraints, and routes tx to that shard. If no satisfying shard exists, the transaction is routed to the DS committee.

Executing sharded transactions. Transactions are applied to a contract in *epochs*, so all nodes in all shards share *the same* contract state at each epoch's beginning and end. Upon receiving a transaction that invokes a transition $\mathcal{T}$ of C with inputs x, a node in a shard $\mathcal{S}$ executes it by applying the transition's logic $\mathcal{F}_{\mathcal{T},x}$ to the contract state σ at hand.

To understand why this is correct and leads to deterministic parallelism, let us think of a "logical split" of the state σ using the per-field join operations $\uplus = \overline{\uplus}_\mathsf{f}$ from the contract's sharding signature. Here, we consider the contract state σ as a join of the two states $\sigma^{\mathcal{S}}_{\text{self}}$, and $\sigma^{\mathcal{S}}_{\text{other}}$ [47], where $\sigma^{\mathcal{S}}_{\text{self}}$ is *owned* by the node's shard $\mathcal{S}$, while $\sigma^{\mathcal{S}}_{\text{other}}$ is not owned, and, hence will not be affected by executing $\mathcal{F}_{\mathcal{T},x}$. Therefore, by running the contract's transition, *i.e.*, executing $\mathcal{F}_{\mathcal{T},x}(\sigma)$, the node obtains the state $\sigma'^{\mathcal{S}} = \sigma'^{\mathcal{S}}_{\text{self}} \uplus \sigma^{\mathcal{S}}_{\text{other}}$, where only the owned part has been changed from $\sigma^{\mathcal{S}}_{\text{self}}$ to $\sigma'^{\mathcal{S}}_{\text{self}}$. Due to the *locality* of $\mathcal{F}_{\mathcal{T},x}$ with respect to the state owned by the node's shard (which is ensured by dispatching the transaction via the ownership constraints), all changes done by the nodes of the shard $\mathcal{S}$ to the state of the contract can be accumulated into the shard-specific state delta $\sigma^{\mathcal{S}}_{\Delta}$, which only includes affected parts of the contract's state that are "logically disjoint" from any changes other shards have made concurrently.

The final state delta $\sigma^{\mathcal{S}}_{\Delta}$ is sent to the DS committee that combines the initial contract state σ with all the state deltas by executing a *three-way merge* [38, 46] using the per-field operations $\overline{\uplus}_\mathsf{f}$ from the signature. This deterministic merge *always succeeds*, thanks to logical disjointness of the state deltas from different shards. Finally, the DS committee executes all the remaining potentially conflicting transactions, obtaining the final contract state, which is broadcast system-wide to all other shards at the end of the epoch.

5 Evaluation

We implemented CoSplit in OCaml as a pluggable checker [20], adding an optional phase to the existing Scilla type checking pipeline. Put together, the analysis, query solver, transaction dispatcher, and state delta merger measure 2900 lines of OCaml code. All interaction between CoSplit and the nodes of Zilliqa network happens via JSON-RPC; that is, the approach can be reused by any other system that provides a way to serialise/deserialise the state of Scilla contracts.

In our evaluation of CoSplit, we focus on two aspects of the tool: the quality of the analysis for sharding signatures (Sec. 5.1) and the impact of using the signatures to the system throughput when executing transactions to popular contracts (Sec. 5.2) from the Zilliqa blockchain.

5.1 Evaluating the Analysis

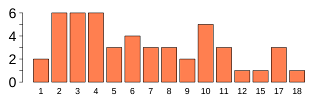
We are interested in quantitative answers to the following questions with regards to the CoSplit analysis:

- Is the analysis fast enough to be used for validating contract-deploying transactions (Sec. 5.1.1)?
- Is the analysis capable of inferring non-trivial sharding signatures for real-world contracts (Sec. 5.1.2)?

To answer these questions, we have run the analysis on all 49 unique contracts from Zilliqa mainnet and testnet.

5.1.1 Analysis performance. All Scilla contracts, upon deployment to the blockchain, are validated by the miners that are forced to parse their code and run the type-checker. We ran the contract deployment pipeline (parsing, type-checking, and sharding analysis) for the contracts in our sample and measured how much time is spent in each deployment stage. Fig. 12 shows the performance breakdown of those three components. The numbers were obtained on a PC with an Intel Core i7-8665U CPU by executing the pipeline for each contract a thousand times and averaging the resulting time for each stage. The analysis adds a significant but acceptable overhead of around 46% to the total contract deployment time. Since contract deployments form a very small fraction of all transactions and happen only once during the lifetime of a contract, the analysis can be run by the miners without affecting overall throughput.

5.1.2 Analysis efficacy. The bar chart on the right summarises the number of transitions (from 1 to 18) for our 49 contracts. While it may be more likely that a contract with a large number of transitions can be sharded efficiently, having many transitions might also indicate having complex logic, making it difficult to infer a useful signature. To quantify the efficacy of the analysis, we introduce some new terminology.



Definition 5.1 (Hogged fields)**.** A contract's transition $\mathcal{T}$ *hogs* a field f in a sharding signature *sg iff sg*'s ownership constraints require a shard to *fully* own f to execute $\mathcal{T}$.

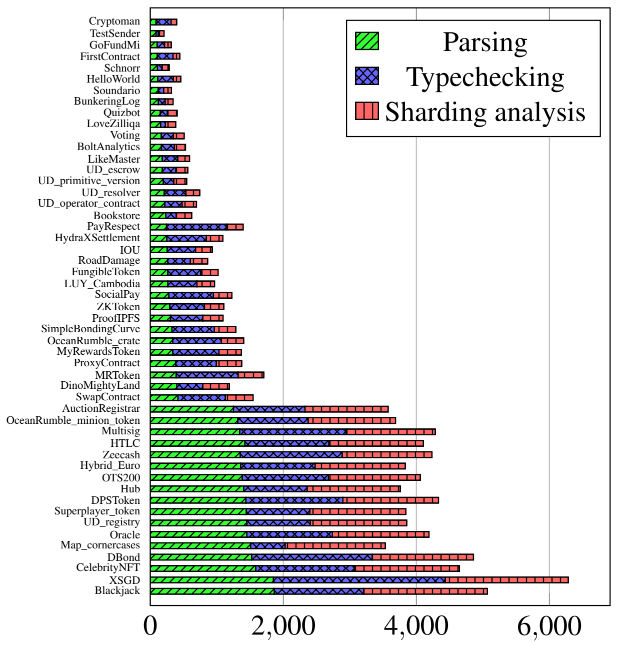


Figure 12. Parsing, type checking, and analysis times (μs).

Definition 5.2. A sharding signature *sg* is *good enough* (GE) for its selection of k contract transitions, *iff* either

- $k = 1$ and the selected transition does not hog fields, or
- $k > 1$ and any field is hogged by at most one transition.

Intuitively, a sharding signature is good enough if it allows for the existence of a contract state, in which some k of the transitions *can be* run in parallel by different shards. Fig. 13a shows the sizes of the largest good enough signatures for our contract selection. It is worth noting that a larger GE signature (in terms of a number k) might perform worse under real-world load than one with a smaller k, which shards different but *more frequently used* transitions. The following definition outlines the signatures worth comparing.

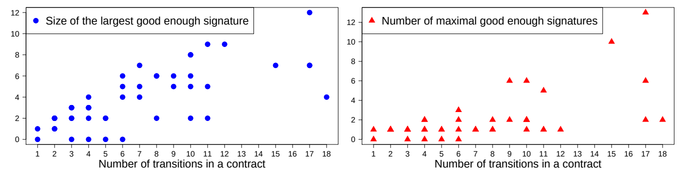
Definition 5.3 (Maximal GE signature)**.** A GE sharding signature is *maximal* if its selection of transitions is not a proper subset of some other GE signature's selection.

A contract might have a number of maximal signatures of various sizes. The plot in Fig. 13b depicts those numbers for our contracts. Computing the maximal signatures at the mining time is impractical, as it requires making $\sum_{k=1}^{n} \binom{n}{k}$ queries to the sharding solver (Fig. 11). Luckily, this computation can be done offline by a contract implementer, who decides prior to the deployment which of the signatures to propose. The miners need to validate only that one signature.

These findings suggest that CoSplit indeed uncovers many opportunities for parallel execution of smart contracts.

5.2 Evaluating Sharded Executions

We evaluate the integration of CoSplit with Zilliqa by measuring the impact on throughput for five representative Scilla



(a) Largest good enough signatures **(b)** Maximal GE signatures

Figure 13. Statistics for Good Enough signatures.

contracts. The chosen contracts are: (1) the most popular contracts on Zilliqa (*e.g.*, UD), (2) equivalent to their popular counterparts on Ethereum currently (*e.g.*, FungibleToken and NonfungibleToken) and in the past (*e.g.*, Crowdfunding).

Contract	LOC	#Trans	Larg.GES	#Max.GES
FungibleToken	439	10	6	2
Crowdfunding	186	3	2	1
NonfungibleToken	288	5	3	2
ProofIPFS	289	10	8	2
UD Registry	500	11	6	2

The table above summarises contract sizes, number of transitions, largest GE strategy, and the number of maximal GE signatures. We set out to answer two main questions:

- What throughput improvement, in terms of transactions per second, can CoSplit help to achieve (Sec. 5.2.1)?
- What is the impact of the overheads imposed by CoSplit-enabled sharding (Sec. 5.2.2)?

Experimental setup. To obtain the throughput figures, we deployed small-scale testnets in various configurations on Amazon EC2 containers. Each node runs on a *t2.medium* machine with 2 logical CPUs and 4GB of RAM, running Ubuntu 16.04. These specifications reflect the minimum requirements needed to run a Zilliqa node. For our benchmarks, we fix the shard size to be 5 nodes per shard and measure the effect on throughput of increasing the number of shards. We use the same shard and DS gas limits as the Zilliqa mainnet.

Selection of sharding signatures. We deploy each of the five contracts in two configurations, one with no sharding information (baseline), and one with a "reasonable" sharding signature, informed by expected usage of the contract. For our experiments, we make the choices as follows:

- For FungibleToken (FT, Zilliqa's ERC20), we shard the `Mint`, `Transfer` and `TransferFrom`, but not `IncreaseAllowance`, `Burn`, or other administrative transitions.
- For NonfungibleToken (NFT, Zilliqa's ERC-721 [61]), we shard `Mint` and `Transfer` (which includes transfer-from functionality), but not `Burn` and `Approve`.
- For ProofIPFS, we shard the transition that notarises a hash, but not the one that removes it from the contract.
- For the Unstoppable Domains (UD) registry, we shard granting a new domain name and updating the record associated with a name, but not transfers of ownership.

For the Crowdfunding contract, there is only one possible choice, which is to shard the `Donate` and `Claimback` (if the goal was not reached) transitions. We argue that our choices reflect what a reasonable contract deployer would select and, as such, the measured throughput reflects probable scenarios.

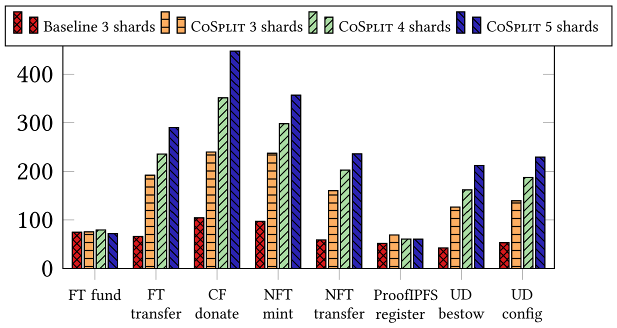


Figure 14. Average TPS for different contract transitions as a function of number of shards, over 10 epochs.

5.2.1 Measuring throughput. After deployment, we subject the contracts, in sequence, to different workloads sustained over 10 epochs (roughly 8.5 minutes) and measure the resulting throughput. As Fig. 14 shows, for most of the workloads in the benchmark, we obtain an almost linear TPS increase as the number of shards goes up. The two exceptions are the "FT fund" and the "ProofIPFS register" workloads. The former transfers fungible tokens from a single source to multiple destinations (all transactions go to the source's shard). The latter notarises a hash, but also keeps a list of notarised items for each user, and thus accesses *two* separate fields, which typically will be *owned by different shards*, hence many transactions need to be processed by the DS Committee. We note that for workloads that do not shard well, performance does not degrade as compared to the baseline, and in some cases (*e.g.* ProofIPFS) marginally improves.

The "FT transfer" workload sends tokens from random sources to random destinations. In the baseline configuration, the throughput is the same as for the single-source workload. The CoSplit-empowered sharding strategy, on the other hand, fully utilises the shards' processing capacity and we get an almost linear increase in throughput as the number of shards increases. A similar effect arises for crowdfund donations. Interestingly, the "NFT mint" workload (which creates new tokens) is also single-source, just like "FT fund". However, the relevant transition does not affect state depending on the identity of the transaction sender, but only on the identifier of the created token. As such, we can obtain linear scaling even for a single-source workload. This is only possible because of the changes to the account-based model that we detailed in Sec. 4.2. Finally, Unstoppable Domains is the most popular smart contract on the Zilliqa mainnet, accounting for over half of the smart contract executions. We manage to shard the most popular transitions on this contract, which account for 90% of usage, and show linear increases in throughput for them as well.

5.2.2 Introduced overheads. Integrating CoSplit adds overheads to transaction dispatch and state delta merging.

Concretely, we see transaction dispatch time increase from an average 8 µs to an average 475 µs, and the state delta merging increasing from 0.8 µs to 48.65 µs per changed state field. This amounts to a 60x slowdown of these operations, most of it as a result of serialisation and deserialisation costs, but this is fully justified by the resulting increase in *overall system throughput*. The overall performance gain comes from the fact that applying a delta is much faster than executing all the transactions that resulted in it. For instance, for FungibleToken, the effects of 50 seconds of transaction execution time can be merged in roughly 0.5 seconds.

6 Discussion and Future Work

Handling integer overflows. CoSplit's signature inference does not take possible integer overflows into account. Overflows (and underflows) may cause a problem in the case when IntMerge is used to join state deltas from different shards that individually do not cause an overflow, but do so when joined. At the moment, our implementation eschews this issue by using unbounded integers, which impose almost no overhead in practice. In the case of bounded integer implementations, a possible solution is to modify the Scilla interpreter, providing it with information about the number N of shards. Specifically, it should perform additional post-hoc validation of a transaction, which will fail if the difference between the initial (per epoch) value v of any affected integer-valued component and its value after the transaction is executed exceeds $\lfloor \frac{\texttt{MAX\_INT}-v}{N} \rfloor$. The information about such components is available in the sharding signatures.

CoSplit and other blockchains. As demonstrated in Sec. 3, the core analysis of CoSplit does not rely on any specific features of Scilla but is easy to implement for it due to the language's minimalism and restrictions (*e.g.*, very limited set of side effects). Should other *account-based* blockchains, *e.g.*, Tezos [30] and Ethereum [69], provide a sharded architecture in the future, we believe, a similar analysis could be implemented for them as well. The key challenge of developing CoSplit for Michelson, the language of Tezos [66], is in reasoning about its stack-based executions, tracking the provenance and cardinality of pushed and popped values. Ethereum's EVM could be supported through decompilation into a high-level language [31]. In order to make such analyses practical for mining-time usage, they will most likely have to be restricted to contracts with no external calls.

7 Related Work

Sharding contracts in blockchains. Several industry proposals outline approaches for smart contract sharding, yet none of them provide an efficient solution for sharding same-contract transactions. For example, the Elrond protocol moves a smart contract to the same shard where its static dependencies lie [25], which takes at least 5 rounds of consensus for a transaction to be finalised. Harmony [32] allows one to deploy contracts in individual shards, with no cross-shard communication allowed. Ethereum 2.0 proposes a cross-shard yanking scheme where the contract code and data is moved into a shard at runtime [68]. The shard then locks the contract to block any parallel execution of other transactions affecting it. At the time of writing, none of these solutions appear to be fully implemented.

The Chainspace protocol allows for sharded execution of smart contracts (as well as state sharding) by representing state evolution as a directed acyclic graph [1], similar to Bitcoin's UTxO transaction model [63]. In Chainspace, contracts have to be written in a specific fashion, so their execution happens *off the chain*. For an unspecified kind of transaction, the authors of Chainspace report linearly increasing throughput of approximately 75 TPS per each two shards of four nodes in each [1, Fig. 6]. Importantly, Chainspace does not address the scalability problem with same-contract transactions (which we do): under high contention for the same contract the rate of aborted transactions rises.

Smart contracts and concurrency. The work by Dickerson *et al.* [19] describes a commutativity-based approach to execute smart contracts by miners *locally* in parallel, using software transactional memory techniques [33, 34]. Unlike our approach that detects possible transaction conflicts pessimistically by means of a static analysis prior to the contract deployment, the work by Dickerson *et al.* exercises the optimistic approach, where the conflicts are identified on-the-fly by the miners, with the corresponding executions aborted. It is not clear how to apply these ideas from optimistic concurrency for efficient sharding without introducing mechanisms for rolling back conflicts at the protocol level.

Recent work by Bartoletti *et al.* makes an observation similar to ours that commutativity (wihch they dub *swappability*) of transactions manipulating Ethereum contracts enables their parallel execution [6]. Their syntactic criterion for inferring swappability is, however, more coarse-grained than our analysis, and is based on determining disjointness of transaction footprints, without taking into account commutativity of operations such as addition. As such, their approach would not allow to shard individual non-conflicting updates in an ERC20 contract as we do. Bartoletti *et al.*'s approach has not been implemented in practice.

Inferring commutativity. Reasoning about commutativity between program parts is an important problem with applications including parallelising compilers [36, 41, 58], speculative execution [33], and race detection [21]. Most of existing techniques for inferring commutativity are based on analysing dynamic executions [2, 21, 28, 67] or by solving SMT constraints [5] and, thus, cannot be used efficiently as a part of transaction validation. Our analysis is close in spirit to the work (in [58]) on static analysis to determine operation commutativity for compile-time parallelisation. Our analysis uses simpler abstractions, allowing it to be implemented in a compositional fashion and have linear execution cost.

References

[1] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A Sharded Smart Contracts Platform. In *NDSS*. The Internet Society.

[2] Farhana Aleen and Nathan Clark. 2009. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *ASPLOS*. ACM, 241–252.

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*. ACM, 487–498.

[4] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the Age of Blockchains. In *1st ACM Conference on Advances in Financial Technologies, (AFT)*. ACM, 183–198.

[5] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *TACAS (LNCS, Vol. 10805)*. Springer, 115–132.

[6] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. 2020. A True Concurrent Model of Smart Contracts Executions. In *COORDINATION (LNCS, Vol. 12134)*. Springer, 243–260.

[7] Bitcoin Wiki. 2019. Scalability. https://en.bitcoin.it/wiki/Scalability, accessed on May 12, 2020.

[8] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. 2010. Concurrent programming with revisions and isolation types. In *OOPSLA*. ACM, 691–707.

[9] Sebastian Burckhardt and Daan Leijen. 2011. Semantics of Concurrent Revisions. In *ESOP (LNCS, Vol. 6602)*. Springer, 116–135.

[10] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66.

[11] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *LICS*. IEEE Computer Society, 366–378.

[12] Lásaro J. Camargos, Rodrigo Schmidt, and Fernando Pedone. 2007. Multicoordinated Paxos. In *PODC*. ACM, 316–317.

[13] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association, 173–186.

[14] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2013. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*. ACM, 1–17.

[15] ConsenSys. 2018. The Inside Story of the CryptoKitties Congestion Crisis.

[16] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

[17] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. ACM Press, 269–282.

[18] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *SIGMOD Conference*. ACM, 123–140.

[19] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding Concurrency to Smart Contracts. In *PODC*. ACM, 303–312.

[20] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muslu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *ICSE*. ACM, 681–690.

[21] Dimitar Dimitrov, Veselin Raychev, Martin T. Vechev, and Eric Koskinen. 2014. Commutativity race detection. In *PLDI*. ACM, 305–315.

[22] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *POPL*. ACM, 287–300.

[23] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[24] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *POPL*. ACM, 2–15.

[25] The Elrond Team. 2019. Elrond: A Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake. https://elrond.com/assets/files/elrond-whitepaper.pdf.

[26] Ethereum Foundation. 2018. ERC20 Token Standard. https://en.bitcoinwiki.org/wiki/ERC20 Online; accessed 4 March 2020.

[27] Ethereum Foundation. 2018. Solidity Documentation. http://solidity.readthedocs.io.

[28] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning Commutativity Specifications. In *CAV (LNCS, Vol. 9206)*. Springer, 307–323.

[29] Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'État. Université de Paris VII, Paris, France.

[30] L.M. Goodman. 2014. Tezos—a Self-Amending Crypto-Ledger. White paper.

[31] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In *ICSE*. IEEE / ACM, 1176–1186.

[32] Harmony Team. 2018. Harmony: Technical Whitepaper. https://harmony.one/pdf/whitepaper.pdf.

[33] Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP*. ACM, 207–216.

[34] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *PODC*. ACM, 92–101.

[35] Sungjae Hwang and Sukyoung Ryu. 2020. Gap between Theory and Practice: An Empirical Study of Security Patches in Solidity. In *ICSE*. ACM.

[36] Guy L. Steele Jr. 1990. Making Asynchronous Parallelism Safe for the World. In *POPL*. ACM Press, 218–231.

[37] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, 637–650.

[38] Gowtham Kaki, Swarn Priya, K. C. Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 154:1–154:29.

[39] 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 *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 583–598.

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

[41] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *PLDI*. ACM, 211–222.

[42] Amrit Kumar. 2018. Provisioning Sharding for Smart Contracts: A Design for Zilliqa.

[43] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze after writing: quasi-deterministic parallel programming with LVars. In *POPL*. ACM, 257–270.

[44] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[45] Leslie Lamport. 2005. *Generalized Consensus and Paxos.* Technical Report 33. Microsoft Research.

[46] Daan Leijen, Manuel Fähndrich, and Sebastian Burckhardt. 2011. Prettier concurrency: purely functional concurrent revisions. In *Haskell*

Symposium. ACM, 83–94.

[47] Ruy Ley-Wild and Aleksandar Nanevski. 2013. Subjective auxiliary state for coarse-grained concurrency. In *POPL*. ACM, 561–574.

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

[49] Nancy A. Lynch and Mark R. Tuttle. 1989. An Introduction to Input/Output Automata. *CWI Quarterly* 2 (1989), 219–246.

[50] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *SOSP*. ACM, 358–372.

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

[52] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer.

[53] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

[54] Ontology Research Team. 2019. Ontology Sharding, Draft v0.2. https://github.com/ontio/documentation/blob/master/sharding/ontology-sharding.pdf.

[55] Manan Patel. 2019. Ethereum Series – Understanding Nonce. Blog post available at https://medium.com/swlh/ethereum-series-understanding-nonce-3858194b39bf.

[56] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. 2011. Commutative set: a language extension for implicit parallel programming. In *PLDI*. ACM, 1–11.

[57] John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium (LNCS, Vol. 19)*. Springer, 408–423.

[58] Martin C. Rinard and Pedro C. Diniz. 1997. Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers. *ACM Trans. Program. Lang. Syst.* 19, 6 (1997), 942–991.

[59] Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *PACMPL* 3, OOPSLA (2019), 185:1–185:30.

[60] Ilya Sergey, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Modular, higher-order cardinality analysis in theory and practice. In *POPL*. ACM, 335–348.

[61] Dieter Shirley. 2018. ERC-721 contract specification. http://erc721.org/.

[62] Alex Skidanov and Illia Polosukhin. 2019. Nightshade: Near Protocol Sharding Design. https://near.org/downloads/Nightshade.pdf.

[63] Flora Sun. 2018. UTXO vs Account/Balance Model. Online blog post, available at https://medium.com/@sunflora98/utxo-vs-account-balance-model-5e6470f4e0cf.

[64] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. 2016. Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 526–545.

[65] Nick Szabo. 1994. Smart Contracts. Online manuscript.

[66] Tezos Foundation. 2018. Michelson: the language of Smart Contracts in Tezos. https://tezos.gitlab.io/whitedoc/michelson.html Online; accessed 4 March 2020.

[67] Omer Tripp, Roman Manevich, John Field, and Mooly Sagiv. 2012. JANUS: exploiting parallelism via hindsight. In *PLDI*. ACM, 145–156.

[68] Ethereum Wiki. 2020. On Sharding Blockchains.

[69] Gavin Wood. 2014. Ethereum: A Secure Decentralized Generalised Transaction Ledger.

[70] Shale Xiong, Pedro da Rocha Pinto, Gian Ntzik, and Philippa Gardner. 2017. Abstract Specifications for Concurrent Maps. In *ESOP (LNCS, Vol. 10201)*. Springer, 964–990.

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

[72] Zilliqa Team. 2017. The Zilliqa Technical Whitepaper. Available at https://docs.zilliqa.com/whitepaper.pdf.