YaleNUSCollege

Towards Locally-Parallel Processing of Smart Contract Transactions

Nicholas Chin Jian Wei

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Dr. Ilya Sergey

AY 2020/2021

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

[✓ Unrestricted access]
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____________ (mm/yyyy) to _______________ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
____
____

Nicholas Chin Jian Wei, Cendana College
Name & Residential College of Student

[illegible]
Signature of Student

02 April 2021
Date

Prof. Ilya Sergey [illegible]
Name & Signature of Supervisor

04 April 2021
Date

Acknowledgements

The following people have provided endless support and guidance throughout my experience in Yale-NUS College, for which I am thankful.

Dr. Ilya Sergey, my capstone supervisor, provided the motivation and guidance throughout my education. His lectures structured my foundational knowledge and inspired my pursuit in my career.

George Pîrlea for the support and troubleshooting efforts throughout this project. His thorough understanding of the Zilliqa blockchain helped me plan and design the new implementation, as well as breezing through debugging erroneous code.

My parents for supporting my education and always taking care of my well-being, and my siblings for being wonderful.

Finally, Courtney and my amazing friends—Allie, Raja, Shaf, Sid—for being great people, endless support, and creating many unforgettable experiences throughout the last four years.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Towards Locally-Parallel Processing of Smart Contract Transactions

by Nicholas CHIN Jian Wei

Blockchain protocols are notorious for poor scalability and low transaction processing throughput. This project proposes a method of locally-parallel processing smart contract transactions for higher throughput. Based on contract static analysis, this project exploits disjoint memory footprints for a lock-free, non-conflicting transaction execution. This paper describes integrating a procedure for local concurrent processing in the Zilliqa blockchain, a sharded blockchain with smart contract support, and evaluates the performance in a distributed network.

Keywords: Blockchain, Concurrent, Smart Contract

Contents

List of Figures

Chapter 1

Introduction

Blockchains have gained significant popularity and importance in providing a decentralized and distributed ledger. Even more so with the adoption of blockchain-based smart contracts, expanding the usages of blockchains beyond purely a decentralized ledger. With blockchains being reliant on a consensus protocol, work must be distributed to and executed on every network on the node. For this reason, blockchains are slow in processing a large amount of transactions [1].

Transactions in blockchains are currently executed sequentially. Sequential execution guarantees all nodes reaching a consensus. However, blockchains fail to utilize the power of modern multi-core systems and multi-threaded, concurrent execution. Ideally, transactions can be grouped together for parallel execution. By exploiting modern computer architecture, blockchains can achieve higher throughput.

This project aims to add concurrency to smart contract executions in efforts to achieve better performance while maintaining the semantics of the transactions and preserving the blockchain protocol.

On Blockchain Efficiency. Blockchains are slow. To achieve a consensus across the network, every node must process the same set of transactions. Now one might suggest that adding additional nodes to the network should improve the throughput. Intuitively, more nodes available for processing equates to more total processing power, hence improved performance.

However, the consensus protocol requires every node to process the same amount of work. So despite adding more nodes, the work is not split and shared across the network. Thus the network will see the same performance or even worse performance due to bandwidth limitations.

Other proposals on improving blockchain performance are also very attractive. Adopting the idea of sharding in distributed databases, various networks shard the blockchain networks into smaller sub-networks, each responsible for a subset of the transactions. Networks such as Zilliqa and Ethereum 2.0 utilize network sharding [6, 15]. But this solution focuses on the structure of the network.

This project is interested in increasing performance at the node level *without* altering the network. As mentioned previously, blockchains fail to utilize modern multi-core computer architecture. Striving for concurrent execution will exploit modern computer architecture which is, for the most part, untouched in production network blockchains. However, automatic concurrency is hard.

On Concurrent Execution. While parallel transactions can help, they come with many concerns. Indeed, concurrent executions *should not* alter the semantics of the blockchain and should be equivalent to sequential executions, only faster.

Recent work by Dickerson et al. explores optimistic concurrency of smart contract execution by speculative execution of transactions with execution rollbacks on conflicting transactions [4]. Although this method of concurrency does achieve speedup, it contains major implications on the network. Their work requires additional message passing of execution schedules between miners and validators of the Ethereum network [4]. Therefore, a change in the network's protocol must be made.

Our Goal. This project aims to provide a method of concurrent execution of smart contracts without altering the network protocol. In this way, transactions can be executed concurrently without requiring extra information in message passing while maintaining the semantics of sequential execution. It aims to provide a solution for optional increased performance at the local level. For members of the network who wish to exploit multi-core processors, they can choose to do so. Otherwise remaining on sequential execution will provide the same results, only slower.

Contributions. The contributions of this report are:

- Designing a runtime in which unmodified transactions with smart contracts can be parallelized within the node.
- Investigation of a real-world blockchain infrastructure to assess the feasibility of this design.
- Implementation of the framework for parallel executions.
- Prelimiary experiments of locally-parallel processing transactions.

Outline. This report will be structured as follows:

- **Chapter** 2: Background knowledge on blockchains and current/proposed solutions on improving blockchain transaction processing performance.
- **Chapter** 3: Technical setup on how Zilliqa blockchain processes transactions in tandem with Scilla, a smart contract programming language.
- **Chapter** 4: Discussing details of the current transaction processing implementation and changes for local parallel processing.
- **Chapter** 5: Testing and evaluating changes to the Zilliqa blockchain.
- **Chapter** 6: Discussions on challenges faced and future improvements.

Chapter 2

Background

In this chapter we provide the necessary background on blockchain consensus protocols, outline the main scalability bottlenecks, and survey the existing approaches for increasing blockchain throughput.

2.1 Nakamoto Consensus

The Nakamoto consensus protocol has been instrumental in the development of blockchain technology. At its very essence, every transaction in the network must be verified by members of the network. After processing a group of transactions, each member of the network comes to a conclusion on the resultant state post transactions, with the common majority being the network accepted state [8]. This allows for the transfer of digital currencies, like Bitcoin, without a centralized regulating body. With the adoption of integrating smart contracts [14] in blockchain, blockchains has been in the spotlight for upcoming technologies and applications.

But the Nakamoto consensus is inherently slow and suffers in scalability roadblocks. Bandwidth limitations are reached due to allocation of workloads to all members across network. Performance limitations are reached due to workload processing being required by all members across the network. Indeed, an increase in the number of members joining the network results in more members who must receive and process all transactions, which slows down the entire network [1]. In practice, Bitcoin sits around 7 transactions per second and Ethereum at 15 transactions per second [6, 11].

Efforts in improving performance of blockchains has been a major focus. For any blockchain to be widely adopted by the public, they must reach a level of performance that can compete with centralized, third-party bodies.

2.2 Concurrency for Increasing Throughput

Despite the scalability issues with the Nakamoto consensus, proposals for alleviating scalability bottleneck have been proposed. We can break them down into two categories: global and local solutions. Global solutions alters the structure of the network while local solutions only alters the node-level. Currently, a popular global solution in increasing performance is through network sharding, a strategy borrowed from sharding distributed databases. By partitioning the network into intercommunicating subnets with disjoint responsibilities, throughput performance increases [3].

Instead of all nodes in a network communicating with each other, nodes now form groups, called shards, which process a proportion of all incoming transactions and strictly maintain only intra-shard communication. A designated shard leader (or leaders) would be responsible for necessary inter-shard communication, minimizing the number of messages being sent.

Suppose a blockchain contains n total accounts and therefore a total of n different states to account for. In the traditional setting, each node must manage every account. However, in a sharded setting, each node now only accounts for $\frac{n}{s}$ states, where s is the number of shards in the network. Adding a node to the network would increase the size of a shard, which would slightly decrease the performance of said shard. However, with a sufficient amount of new nodes, a new shard could be created, further partitioning the number of states each node must account for and therefore increasing network performance. Hence, a sharded blockchain offers an improvement on scalability.

A local solution for increase scalability is adding concurrency to transaction processing. Many modern systems are equipped with multi-core processors that allow for efficient concurrent processing. Currently, many (if not all) blockchains process transactions sequentially. Sequential processing is simple in design and easier to trace and debug. Unfortunately, this means modern processors' potentials are not fully utilized.

However, we cannot simply run transaction processing in multiple threads concurrently. Suppose an account had a balance of 100 tokens they would like to transfer out 60 and 70 tokens in two different transactions. In a sequential setting, the node would deny the second transaction as the account would have an insufficient balance. In a concurrent setting, we cannot ensure [1] that either transaction would know the other has already occurred. Indeed, if 2 threads read that the account has a

[1]Without any use of careful granular locking mechanisms

sufficient balance of 100, both transactions would be processed without error despite the account overspending 30 tokens.

One key area where we can add concurrent processing safely is on disjoint states. If a batch of transactions only affect disjoint accounts or states, we could safely process them concurrently. Suppose two different accounts each wanted to transfer tokens to another account, both of which are neither the other sending account nor the same receiving account. It would be safe to process both transactions concurrently as the balance of one sending account should not affect the balance of the other sending account.

Define accounts A, B, C, D as user accounts who all contain some amount of tokens as part of their balance. They can send and receive tokens by transferring to each other. Suppose we have transactions $t_1 = \texttt{transfer}\ (A \rightarrow B, 100)$ and $t_2 = \texttt{transfer}\ (C \rightarrow D, 100)$ where $(A \rightarrow B, 100)$ denotes account A transferring 100 tokens to account B. It is clear that both transactions act on disjoint accounts and states since both transactions do not share accounts. Both such transactions can be parallelizable.

Now suppose we have transactions $t_3 = \texttt{transfer}\ (A \rightarrow C, 100)$ and $t_4 = \texttt{transfer}\ (B \rightarrow C, 100)$. From the perspective of account C, they will be receiving 100 tokens each from accounts A and B. Regardless of which transaction is processed first, the net effect is still $+200$ tokens. This implies both transactions have commutative effects (by exploiting the commutativity of addition), and therefore can be parallelized.

In Figure 2.1, we group how each transaction affects which account states. Transactions 1 and 2, marked in black and red, clearly depict the

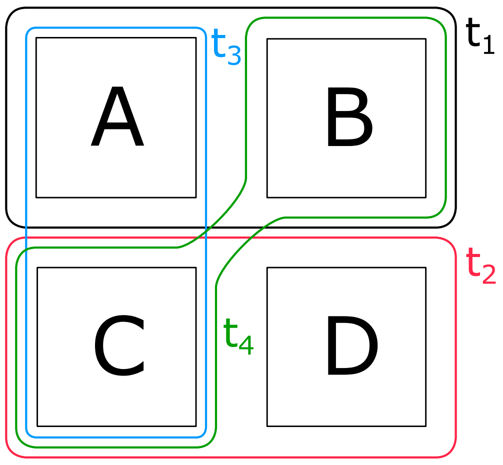

FIGURE 2.1: Grouping of affected account states based on transactions

disjoint account states between both transactions. Transactions 3 and 4, marked in blue and green, show that both transactions act on account *C*, but their commutative effects allow both to be parallelizable.

For simplicity and setting a manageable scope, this report focuses solely on disjoint transactions. Transactions that share some state require much more care to ensure the semantics of the transactions are still maintained.

2.3 Recent Works

Concurrent processing in blockchains is not novel; recent works have suggested multiple approaches to concurrent processing.

Optimistic concurrency. Dickerson et al. implements concurrent execution of smart contracts in the Ethereum blockchain. Their proposal follows a strategy of concurrent execution through optimistic concurrent execution. Dickerson et al. based their tactic on software transactional memory, requiring a record for each read and write on a piece of data. Transactions are executed in parallel and a log of state changes are recorded. Any transactions that may conflict must have their effects rolled back and subsequently executed sequentially [4].

For this reason, nodes in the Ethereum network must be altered to verify correct concurrent executions and subsequently require repeated execution of the same transactions in the event of conflicts. Thus, the Ethereum protocol must be altered. Furthermore, this optimistic approach incurs memory overhead by requiring a transactional log to be kept for every data read and write.

Static analysis on disjoint memory accesses. Baroletti, Galletta, and Murgia propose a theoretical approach to concurrent smart contract processing. They propose that transactions are *swappable* if some transactions $t_1 t_2$ processed in either order produces the same blockchain state [2]. Their work solely focuses on only disjoint states but does not detect commutative operations on smart contracts. As such, they be unable to process ERC-20 [5] based smart contracts, which are predominantly the most commonly executed smart contract [9].

Empirical study on historic transactions. Saraph and Herlihy performed an empirical study on historical transactions of the Ethereum blockchain. Their work discovered that optimistically running transactions concurrently encountered very minimal conflicts that required rollbacks and their simplest strategies still produced non-trivial speedup. Notably, speedups began to declined as the volume of transactions increased [10]. Handling the volume of transactions can be solved through network sharding, which Ethereum[2] does not support.

Static analysis for sharding smart contracts. Kumar, Pîrlea, and Sergey propose a static analysis, named *CoSplit*, for sharding smart contracts. Similar to sharding a blockchain network, they show that smart contracts themselves can be sharded by their memory footprint via static analysis. This footprint allows different parts of the contract to be partitioned across the network, increasing performance in a linear relation to the number of shards of the blockchain network [7]. CoSplit is currently implemented in a development branch of Scilla (see Chapter 3) and a development branch of the Zilliqa blockchain. Furthermore, the static analysis can dually be used to determine safe concurrent processing of smart contract transactions.

With the findings from CoSplit, we can build on sharding smart contacts and utilize the static analysis for concurrent execution. Zilliqa provides support for smart contracts through the Scilla programming language. As such, we decided to proceed with exploring Zilliqa and modifying it for concurrent processing.

[2]Ethereum 1.0 does not support sharding but Ethereum 2.0 (currently in development) supports sharding [6]

Chapter 3

Technical Setup

In this chapter, we explore the technical setup of the Zilliqa blockchain, Scilla smart contract programming language, and the intricacies of the current implementation.

We are interested in how Zilliqa processes transactions that invoke smart contracts. We explore how Zilliqa processes such transactions and note how we may modify it for concurrent processing.

3.1 Zilliqa Blockchain

Zilliqa is the first sharded blockchain and supports smart contracts [15]. As previously discussed, a sharded blockchain partitions its nodes into subnets that are directly responsible for their own disjoint state. In this way, Zilliqa scales well as nodes are added and as the number of transactions increase.

Concretely, Zilliqa contains three main groups: the shards, the Directory Service (DS) committee, and the lookup nodes. Lookup nodes distribute incoming transactions to the shards and DS committee. Shards consist of a group of nodes that are responsible of their own unique partition of the global state. Shards process transactions that are assigned to them. The DS committee is a special shard that communicates with all other shards. They are assigned transactions that alter states amongst multiple shards.

Looking at a round of transaction processing, named an epoch, we can understand how the entire network progresses and maintains a shared global state. We can break down each epoch into 3 phases.

In the first phase, transactions are sent to the lookup nodes. The lookup nodes then distribute transactions based on their assigned shard. Those transactions that would affect multiple shards are sent to the DS committee.

The second phase consists of processing transactions and forming micro-blocks. Each node begins to process their assigned transactions. Then, the shards undergo a round of practical byzantine fault tolerance (PBFT) consensus protocol to arrive at a consensus of a new shared state. Once a consensus is reached, a micro-block is formed.

The thrid and final phase consists of forming the final-block. At this stage, all shards would have already formed their respective micro-blocks. These blocks are sent to the DS committee. The DS committee undergoes PBFT protocol to arrive at a consensus of which micro-blocks to account for. Once a consensus is reached, a final-block is produced, containing all state changes of each micro-block. This final-block is then added on the chain of blocks, extending the blockchain. Finally, the final-block is then broadcast to the shards so that each node is aware of the new global state, ending the epoch [15].

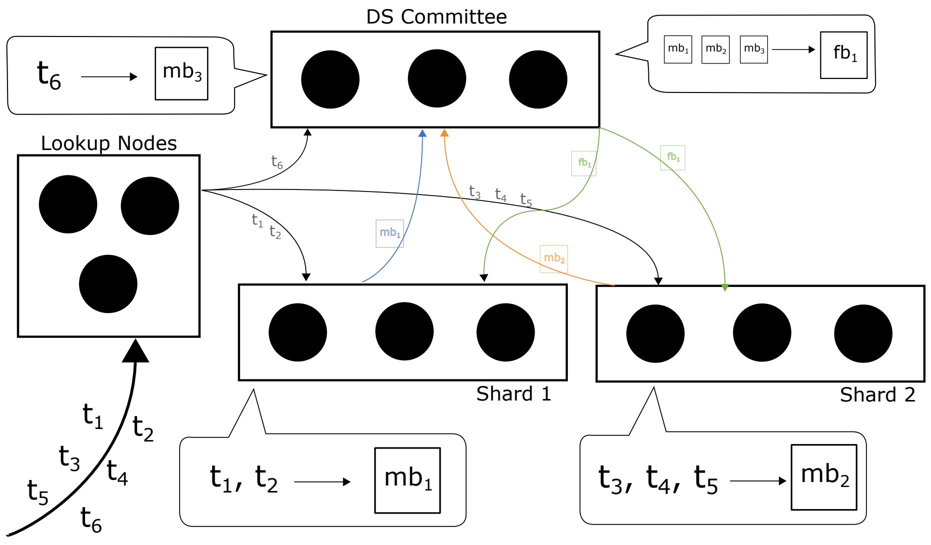
DS Committee
$t_6 \longrightarrow mb_3$
mb_1 mb_2 mb_3 $\longrightarrow fb_1$
Lookup Nodes
t_6
t_1 t_2
mb_1
t_3 t_4 t_5
fb_1
fb_1
mb_2
Shard 1
Shard 2
t_1 t_2 t_3 t_4 t_5 t_6
$t_1, t_2 \longrightarrow mb_1$
$t_3, t_4, t_5 \longrightarrow mb_2$

FIGURE 3.1: Zilliqa network and transaction processing

Now that we have an understanding of how Zilliqa maintains a distributed global state, we can now explore Scilla and smart contracts.

3.2 Scilla

Zilliqa currently supports smart contracts written in Scilla. Scilla is an intermediate, functional language designed for programming smart contracts [12, 13]. It is an explicitly-typed language, aimed to ensure safety in programming smart contracts and their transitions. For this report, two key areas of Scilla will be relevant to concurrent execution of smart contracts.

Firstly, Scilla's library consists of only pure functions. In other words, each library function has no side effects (like printing, changing field variables, etc.) and will always return the same output given the same input. For this reason, executing such library functions concurrently will yield the same result. The only area of contention in executing smart contracts concurrently arises from the smart contract's defined functions, in other words, the programmer's code. Such mutable effects can be statically analyzed using CoSplit to determine the areas of memory a function might touch.

Secondly, a smart contract in Scilla is made up of field variables and transitions, or commonly known as functions. The identifier *transitions* is derived from finite state machines. When invoking a transition of a smart contract, the smart contract will transition between one state to another. For instance, invoking `Increment()` in Figure 3.2 will increment the counter in the smart contract.

```
scilla_version 0

contract Counter

(owner : ByStr20)

field counter : Uint128 = Uint128 0

transition Increment()
  c <- counter;
  inc = Uint128 1;
  new_c = builtin add c inc;
  counter := new_c
end
```

FIGURE 3.2: Counter smart contract

Transitions can take in arguments, such as transfer amounts, strings, but more importantly, they can receive addresses. Addresses could be associated with user accounts or even other smart contracts. Such addresses could be used in message passing by the transition. Transactions that invoke transitions who then send messages to other smart contracts are considered multi-call transactions. Multi-call transactions are difficult to determine the safety of concurrent execution. Indeed, we must know how the called smart contract behaves, which by itself might call another contract.

3.3 Implementation Intricacies

Before we begin making changes to the current implementation of transaction processing, we must first understand the current implementation.

We begin at the node level. First, a transaction packet is received by the node. The transactions are then selected one-by-one for verification. Verification checks if the account that invoked the transaction has enough balance to run the transaction, if the account exists, and if the transaction blockchain ID matches. Once verified, the node can begin running the transaction.

Now, the node stores a snapshot of the current account [1] state. As the transaction is being processed, each step takes some amount of gas. Gas acts as a fee for the sending account to pay for the network to process the transaction. If the transaction runs out of gas or runs into an error, the account state is reverted back to the initial snapshot.

Then, the node starts up a local Scilla server. The server will interpret the smart contract's transition and execute the code. Since all the data exists on the blockchain, the node must send the data over to the server for execution. The smart contract is serialized and deserialized through message passing via inter-process communication (IPC). While the Scilla

[1]For smart contract call transactions, this would be the current smart contract state

server interprets the executes the transition, any fields of data read/written to are communicated to the node to reflect any changes in the smart contract, via IPC. Once the transition finishes its execution in Scilla, the server is then shut down.

Finally, at this point, all changes to the account would have been reflected and the transaction finishes processing. The node continues to process any other transaction within the packet until either the gas limit is consumed, the default transaction processing timeout is triggered, or all transactions in the packet is processed.

After all transactions are processed, the node will create a micro-block and match it against (or propose if leader) the shard leader's proposed micro-block. Once a consensus is reached through the PBFT protocol, the shard leader will then broadcast the micro-block to the DS committee.

From this starting point, we will begin integrating sharding analysis provided by CoSplit. At the point of receiving a transaction calling a smart contract, CoSplit can perform static analysis to provide a memory footprint of the of transition being called. The memory footprint provides us important details, notably which transitions are single or multi-call transitions and which data fields are accessed by the transition and how they are accessed.

We will focus on only single-call transitions as multi-call transitions are much harder to determine concurrent execution safety. Next, we will use the static analysis for sorting paralellizable transactions based on the memory footprint. Those with commutative effects or disjoint memory accesses from other transactions are good candidates for concurrent execution.

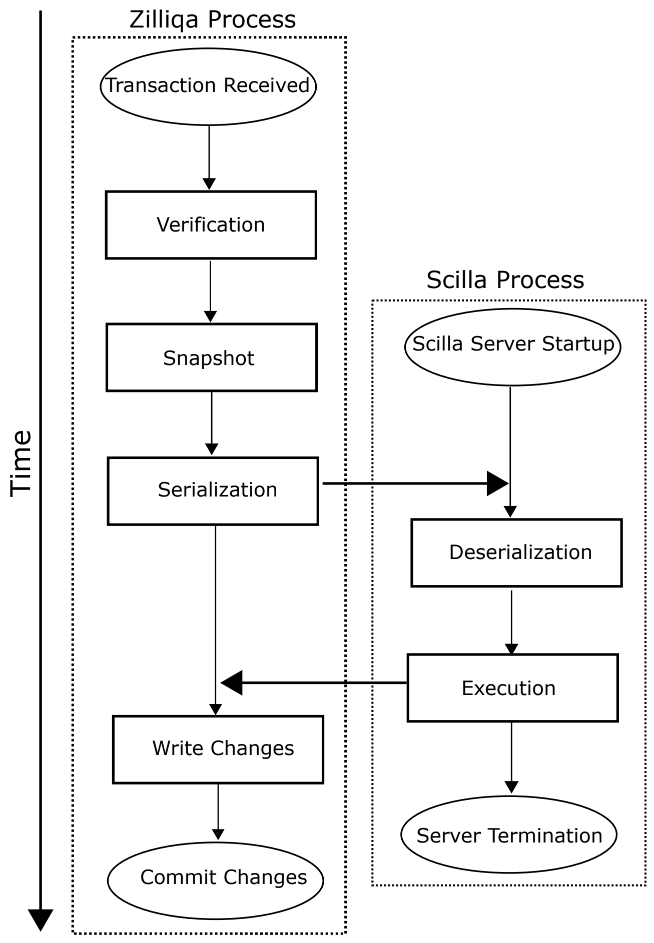

FIGURE 3.3: Transaction Processing IPC

With the knowledge on how Zilliqa processes transactions and executes smart contracts, we can begin exploring the necessary changes for concurrent execution. We will have to modify how Zilliqa sorts transactions, how Zilliqa communicates with Scilla, and how Zilliqa nodes arrive at a consensus.

Chapter 4

Incorporating Concurrency

In this chapter, we go through an outline of what implementation changes needed to be made in order to achieve concurrent execution. For each point, we expand the detailed changes and discuss the success, drawbacks, and failures of each change.

4.1 Outline

As briefly discussed in Chapter 3 and seen in Figure 3.3, the sequential transaction processing pipeline follows a few distinct steps. Expanding on those steps, we begin an outline of each portion of the implementation that needs to be changed.

1. Section 4.2: **Transaction selection** in its current implementation is purely based on selecting the highest gas fee from the transaction pool. Nodes will prioritize processing transactions with higher gas fees to reap the larger rewards. In a concurrent setting, we must sort transactions into parallelizable and non-parallelizable transactions.

2. Section 4.3: Mutual exclusion (**mutex**) **locks** were used to protect accounts from being accessed by multiple threads concurrently. When

processing transactions concurrently, we can ignore those locks entirely if all transactions work on disjoint states. In this way, each thread will work on their own disjoint partition of the account state, so no two threads will interfere with each other.

3. Section 4.4: The current implementation **reverts** any **changes** entirely. That is to say, it reverts back to the screenshot prior to any transaction execution. However, in a concurrent setting, multiple threads might be accessing different parts of the account, so the entire account cannot be reverted to the old snapshot provided one thread fails. Instead, we must keep track of which changes each transaction has made and solely revert those changes.

4. Section 4.5: **Verifying** a **micro-block** in a shard currently verifies that each transaction processed matches those of the leader and are processed in the same order. In a concurrent setting, we cannot guarantee that each transaction is processed in the same order. As such, a new way of composing and verifying micro-blocks must be made.

4.2 Transaction Selection

Transaction selection is straightforward: transactions are removed from the pool and added into a priority queue. When the transaction pool is empty, the node removes a transaction from the priority queue and begins executing the transaction. This continues in a loop until no transactions exist or the gas used exceeds the gas limit.

```
void ProcessTransactions(int gas_limit) {
  int gas_used = 0;
  while (gas_used < gas_limit) {
    if (!transaction_pool.isEmpty()) {
      Transaction t = transaction_pool.pop();
      sorted_transactions.insert(t);
    } else if (!sorted_transactions.isEmpty()) {
      verifyAndExecute(sorted_transactions.pop());
    }
  }
}
```

FIGURE 4.1: Simplified `ProcessTransactions()`

For concurrent processing, we must consider a few issues. Firstly, not all transactions can be executed concurrently. Since the report focuses on concurrently executing smart contract transactions, all other transactions (normal crypto transfers and contract deployment) are assumed to be ran sequentially. While these types of transactions could potentially be executed concurrently, we maintain their current execution pathway as it will retain safety and simplicity.

Even after filtering out other transactions, not all smart contract executions can be executed concurrently. Some transitions in smart contracts touch the same pieces of data regardless of who sends the transactions. We are focused on transactions on disjoint states, so we filter these transactions out. Furthermore, we want to avoid multi-call transactions, as CoSplit static analysis is currently unable to determine memory footprints of chained transition calls.

With these two conditions in mind, we can filter transactions into 2 priority queues. The first priority queue contains those transactions that can be ran concurrently. The second contains transactions that must be ran sequentially. From these two priority queues, we perform transactions in batches and switch between each queue.

Suppose we receive too many transactions for a node to process in a single epoch. If we naively process parallelizable transactions first before processing sequential transactions, then the sequential transactions could run into a starvation problem, therefore none of them would be executed. By toggling between both priority queues in batches of some set size, we eliminate the starvation problem.

```
void ProcessTransactions(int gas_limit) {
  Threadpool threadpool(NUM_THREADS);
  int gas_used = 0;
  int batch_size = 32;
  while (gas_used < gas_limit) {
    if (!transaction_pool.isEmpty()) {
      Transaction t = transaction_pool.pop();
      if (t.isContractCall() && t.isNonMultiCall()) {
        con_sorted_txn.insert(t);
      } else {
        seq_sorted_txn.insert(t);
      }
    } else if (!isConExec(batch_size)) {
      verifyAndExecute(seq_sorted_txn.pop());
      txns_processed += 1;
    } else if (isConExec(batch_size)) {
      threadpool.addJob(
        verifyAndExecute(con_sorted_txt.pop());
      txns_processed += 1;
    }
  }
  threadpool.joinAll();
}
```

FIGURE 4.2: Concurrent `ProcessTransactions()`

While this does work for sorting out single-call transactions from other types of transactions, it does not filter out transactions that could be affecting the same pieces of data. In line 7, `t.isContractCall()` and `t.isNonMultiCall()` is a simplification of checking the static analysis. To further improve the sorting, this call should also analyze which data fields the transition would be affecting and if we can execute them concurrently and join using any commutative effects.

For example, suppose a smart contract transition added some `amount` to a map of accounts $\rightarrow$ balance called `balances`. The current check with static analysis properly separates when all accounts calling this transition are unique. What it does not check is if an account is calling the transition twice. Calling it twice would be safe for concurrent execution assuming the transition[1] joins effects through commutativity of addition.

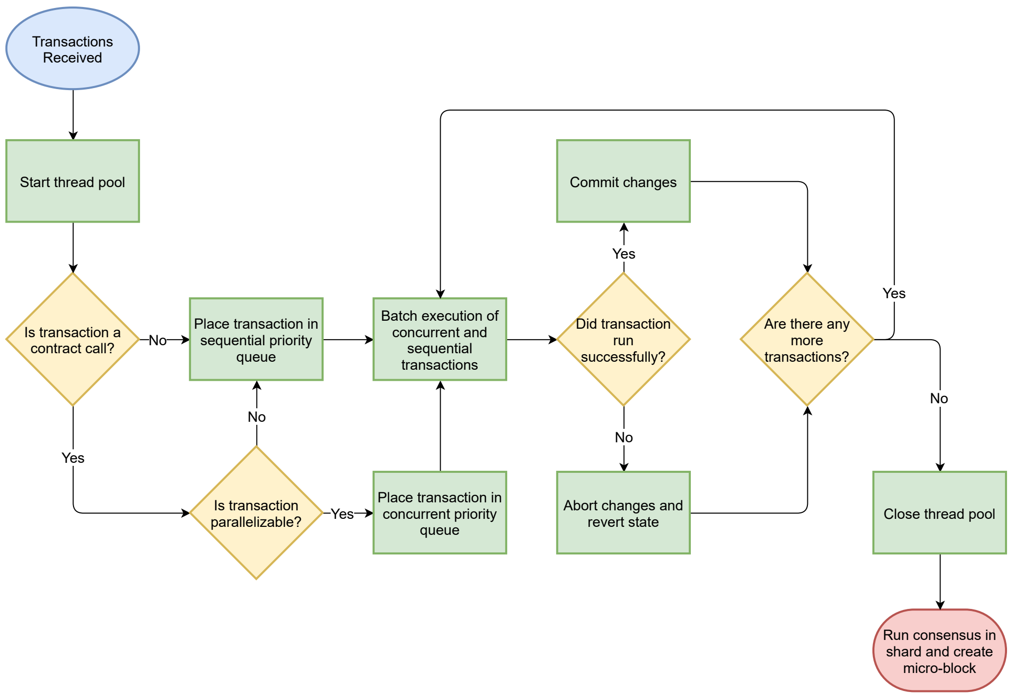
Transactions Received
Start thread pool
Is transaction a contract call?
No
Yes
Place transaction in sequential priority queue
No
Is transaction parallelizable?
Yes
Place transaction in concurrent priority queue
Batch execution of concurrent and sequential transactions
Did transaction run successfully?
Yes
Commit changes
No
Abort changes and revert state
Are there any more transactions?
Yes
No
Close thread pool
Run consensus in shard and create micro-block

FIGURE 4.3: Flowchart of transaction processing in parallel runtime

However, sorting such occurrences could be very expensive. Suppose the `con_sorted_txn` priority queue currently contains transactions from accounts $a_1, a_2, \ldots, a_n$ that all execute on disjoint states of the same smart contract, but cannot be joined through a commutative operation. If we

[1]The transition/contract writer is required to provide the joining mechanism for CoSplit static analysis of the smart contract

find that each account called the same transition through another transaction, we must remove all n transactions from the priority queue and slot all transactions into the sequential priority queue. This would incur a massive overhead proportional to the number of conflicting transactions, which would likely be slower than naively running all transactions sequentially. In such a case, we would require a more clever sorting algorithm.

4.3 Mutex Locks

Mutex locks are currently used to lock entire accounts when processing transactions. This is guaranteed safety as no two threads can access the same account at the same time, ensuring no data races on reading and writing data.

```
unique_lock<shared_timed_mutex>
  g(m_mutexPrimary, defer_lock);
unique_lock<mutex> g2(m_mutexDelta, defer_lock);
lock(g, g2);
```

FIGURE 4.4: Account wide mutex locks

For concurrent execution of transactions, we must remove the locking mechanism. In fact, for disjoint state transactions, locking is unnecessary. Indeed, if two threads modify two different areas of the same account, they will not interfere with each other and thus can safely be executed concurrently.

Even if we ran transactions that affect the same state, if they have some commutative operation to join effects of all transactions, then we again would not need the locking mechanism. Instead, we would exploit the commutative effect of the operation and write the combined change into the account. This would require more bookkeeping and extra care when executing such transactions.

```
if (!CONCURRENT_PROCESSING || !isConcurrent) {
  unique_lock<shared_timed_mutex>
    g(m_mutexPrimary, defer_lock);
  unique_lock<mutex> g2(m_mutexDelta, defer_lock);
  lock(g, g2);
}
```

FIGURE 4.5: Account wide mutex locks with check

We add a node-wide variable `CONCURRENT_PROCESSING` that would retain the locking mechanism if the node chooses to run all transactions sequentially. Concurrent processing is designed to be a user's choice, so we design the optional change to be simply toggle-able through changing a configuration setting. A second flag will be checked if the incoming transaction is set to run concurrently or sequentially. If both values are `true`, we skip the locks entirely.

4.4 Reverting Changes

On the event a transaction runs into an error or the gas limit has been utilized, a node will revert any changes made to the account. Currently, changes are made simply by reverting to an old snapshot taken right before any contract execution. In a sequential setting, this method is safe since there will be no concurrent access to the same account. In a concurrent setting, we must be weary of any committed changes other threads could have made. Indeed, we do not want to overwrite and erase any committed changes. The user who sent the transaction would receive a confirmation that the transaction succeeded, whereas any changes may not have been recorded.

As discussed in Chapter 3, nodes run a local Scilla server in a separate thread for contract execution through IPC. Communication between the server and the node is performed through JSON remote procedure calls (JSON-RPC). Messages are passed between the two processes, executing both reads and writes to the contract state both in the local node and Scilla.

Knowing this, we can intercept any reads to initial values and store them within the final message, containing the receipt of executing the contract in Scilla. When the node analyzes the receipt, we can determine if the contract succeeded or failed. If a success was made, we keep all changes made. Otherwise, we revert all changes made by writing the original values (the read values in the receipt) to the current contract state. In this method, we preserve changes made on disjoint areas within the contract.

Firstly, we must change how revisions are made. `RevertPrevState()` simply overwrites with the previous screenshot. We alter the function such that it receives the contract execution receipt as an input. We then iterate through the receipt's fetched values and insert them back into current contract state (or remove values if none was originally there).

Secondly, we must change the IPC messages between the Scilla server and the node. Specifically, we change how the node handles messages from the Scilla server. During contract execution, the Scilla server will send messages to the node every time a value is read or written in a contract field. For instance, a contract may read the value from a mutable

```
void ContractStorage2::
       RevertPrevState(Json::Value& receipt) {
  for (const auto& key : receipt) {
    if (key.get().empty()) {
      t_stateDataMap.remove(key);
    } else {
      t_stateDataMap.insert(decodeBase64(key.get()));
    }
  }
}
```

FIGURE 4.6: Modified contract reversion

map of user addresses → balances and later write to that same memory location with a new balance.

For this reason, we can intercept these read messages to collect the initial values of the contract pre-execution. These messages could then be collected into a JSON receipt and passed to `RevertPrevState()` for contract reversion, if necessary.

Since Zilliqa stores the contract state as a map from `string` → `bytes`, then we structure the JSON receipt as a dictionary from string to bytes. Each string key maps to a piece of data, including entire mutable structures. Suppose a contract's address is `zil1hgg7...` and the contract has a map named `balances` containing user account balances. Accessing a value in `balances` is performed by appending the addresses and data values with a network defined separator.

```
t_stateDataMap.find("zil1hgg7..." + SCILLA_INDEX_SEPARATOR +
  "balances" + SCILLA_INDEX_SEPARATOR + "0x134a1b...");
```

FIGURE 4.7: Data read from Zilliqa contract state

With these two changes, our receipts will simply be a JSON containing keys of memory locations and values of bytes converted to strings. Any empty string implies that the field value was initially empty prior to contract execution, so those values will be removed. This matches the logic of Figure 4.6 and will properly revert the contract state of disjoint state transactions.

```
{
  "zil1hgg7...0x16isBacker0x160x134a1b..." : "true",
  "zil1hgg7...0x16balances0x160x134a1b..." : ""
}
```

FIGURE 4.8: JSON receipt of read values

We can feed the new receipt into the modified `RevertPrevState()` to revert the specified portions of data for that transaction execution. However, such changes are only limited to transactions with disjoint memory footprints. Had 2 transactions acting on mutual areas of memory been executed with one requiring reversion, we cannot ensure that the succeeding transaction changes are committed. Indeed, this reversion scheme could allow a single transaction to be nullified by overwriting the memory value with the original. Reversion for transactions with commutative effects require much more intricacy, requiring the reversion to perform a negation rather than a value overwrite.

4.5 Micro-block Verification

In a sequential setting, micro-block verification simply checks if the transactions executed are the same as the leader's execution and follow the same execution order. In a concurrent setting, we cannot guarantee that transactions orders are the same. Since we divide our transactions into concurrent and sequential executions, we must devise a new verification method containing both methods of execution.

For the sequential executions, we can continue with checking execution order of a set of transactions. Indeed, we should arrive at the same state shard-wide after executing transactions in the same order. For concurrent executions, we instead should check set equality. Since any permutation of a set of concurrently executed transactions is valid, then set equality will verify that states are in consensus shard-wide.

So, we must also now tag each transaction by their execution method. From this, we separate those executed sequentially and concurrently and verify each set through their respective methods. Additionally, set equality on concurrent transactions will also apply for transactions with commutative effects. Indeed, commutativity implies that transaction ordering does not matter.

With the proposed changes, we can begin implementation and incorporating concurrency into Zilliqa transaction processing. By altering transaction selection and batch processing, account-wide mutex locks for concurrent access, state reversion for failed executions, and micro-block formation and consensus, we can begin testing and evaluating processing disjoint memory footprints.

Chapter 5

Preliminary Evaluation

In this chapter we begin running and evaluating the proposed changes to Zilliqa for parallel processing of smart contract transactions. We first explore the testing framework and environment then evaluate how the network performs.

5.1 Testing Framework

To begin testing our proposed implementation, we must first construct some compatible transactions. The Crowdfunding smart contract (see Appendix A) provides the transition `Donate`. On a successful donation, the smart contract will add the sending account to the field map `backers` and insert their donation amount. As such, `Donate` is a good candidate for our transactions each transaction will have a disjoint state, assuming all transactions have unique accounts.

Before we send these transactions, we first must create and fund accounts in Zilliqa to send the transactions and deploy the smart contract. We create 100 transactions to transfer some amount of Zilliqa's cryptocurrency (ZIL) to each account so they can interact with the smart contract.

Finally, we send a one more transaction to deploy the smart contract to the network. Since all these transactions are not smart contract calls, they will all be executed sequentially. They are merely intended to setup the network state.

Once each account has been funded and the contract deployed, we can begin by sending our transactions to call the Crowdfunding smart contract. We construct 100 transactions, each having a unique sending account, all sending 100 ZIL. By the time all transactions are completed, we expect the `backers` field to contain 100 mappings from accounts $\rightarrow$ amount donated, totalling to $100,000$ ZIL donated.

To compare the performance of concurrent versus sequential processing, we set a timer prior to running the 100 smart contract calls. We stop the timer when we receive a response from the network that all transactions sent out are completed. Doing this both in the concurrent and sequential setting will provide some comparison to measure performance.

5.2 Network Structure

Testing concurrent execution of transactions requires a custom testing network (testnet) to run transactions. Initially, the testnet was hosted on a single machine, with each node represented virtually through a separate process for each node. The network consisted of 21 nodes: 5 nodes will form the DS committee, 15 nodes will form 3 different shards of 5 nodes each, and 1 lookup node. The next step brought testing into a distributed network. Now, each node is its own separate machine. Each node will not interfere with one other and will not compete for the same resources on a single machine.

5.3 Evaluation

Overall, after running tests, there was partial success. The nodes in the network managed to startup different threads and run contracts concurrently. Progress was made and some transactions were processed successfully, however there were many areas of failure causing failed transactions.

Processing Time. Tests showed that processing transactions took much longer than sequential processing. This is not due to the increased overhead of spawning new threads or sorting transactions, but rather due to implementation bugs causing failed transactions, requiring later epochs to retry processing the failed transactions. Nevertheless, sequential processing showed to process all 100 transactions in as little time as 1 minute. Concurrent processing took much longer and sometimes ran indefinitely.[1]

Constructing the Thread Pool. In regular use cases of a thread pool, we construct the thread pool when we want to use it, feed it jobs for processing, and terminate all threads and the thread pool when all jobs are completed. Naively, the proposed implementation does as such. The advantage is that we ensure each transaction is processed entirely before proceeding to verification.

However, repeatedly recreating and tearing down a thread pool is more expensive than maintaining the pool indefinitely. For users who wish to enable concurrent processing, indefinitely maintaining the pool

[1]Tests with only 10 transactions showed 1 minute for sequential processing and 24 minutes for concurrent processing

would be advantageous as they would likely be concurrently processing transactions for future epochs and it would only incur a one-time overhead cost of starting up the pool. While no transactions are being processed, sleeping threads would take minimal space and processing power.

To improve the design, we can create the thread pool as soon as the node is starting up. By running the thread pool on startup, we only incur the thread pool construction overhead once. Threads will sleep until they are called to run transactions. To ensure that all threads complete all of the outstanding workload, we set a counter that atomically increases each time a transaction has been processed. All processing is complete when our counter matches to initial number of transactions received, or transaction processing timeout is reached.

Shared Scilla IPC Handler. The Zilliqa client that handles Scilla IPC is a singleton object. The object will be assigned the address of the contract accesses and executes the transition called. In the sequential setting, this works as only one transaction is being processed at a time. However, in the concurrent setting, multiple threads would be accessing the same object. Concurrent usage can lead to overwrites and loss data. As such, many transactions were failing due to missing or overwritten data.

One possible change is to alter the Zilliqa client so that it can handle multiple accounts. We can create a mapping from transaction ID $\rightarrow$ contract address. This way, each thread will only consume messages that correspond to their current transaction ID. Another way is to spawn multiple instances of this singleton object, so that each thread will be assigned their own, unique client.

Shared usage of Scilla Server. Similar to the previous source of bugs, the Scilla server also caused issues when running concurrently. A single server is started to interact with Zilliqa and process the contract. In a concurrent setting, the server would be responding to multiple messages pertaining to different contract calls. These messages then cause errors as messages intended for one transaction may be consumed by a thread assigned to a different transaction.

Very similar to the Scilla IPC handler, we can either create multiple, local Scilla servers, or restructure the server such that it can handle multiple messages at once. The former is much simpler as each thread can be assigned a unique ID that allows exclusive access to their respective server. The latter requires larger restructuring of how Scilla servers run and the implementation will be quite complex.

Overall, concurrent processing is very much possible, but the implementation is very complex. Many parts of the current implementation require restructuring and major overhauls. Concurrent processing is promising but there will be a lot of necessary future improvements to create a stable system.

Chapter 6

Discussion

6.1 Source Code

The development branch of Zilliqa described in this report is based on Zilliqa 7.2.0. The developmental branch integrates smart contract sharding based on findings from CoSplit. Instructions to run the local testnet can be found in the `README`. To enable concurrent processing, toggle `<CONCURRENT_PROCESSING>` in `constants_local.xml` to `true` before building and running Zilliqa. The source code can be found on `https://github.com/Zilliqa/zilliqa/tree/feature/concurrent_execution`.

6.2 Encountered Issues

This report has shown that concurrently processing transactions is feasible but has a long way till production deployment. Testing for functionality and reliability is, unfortunately, restricted to a distributed setting. In the local tests,[1] a single processor was unable to keep up with the demand of several virtual nodes. A single processor would be shared amongst

[1]Local tests were initially done on a machine with AMD Ryzen 3700X, 16GB 3600MHz RAM, running WSL2

all nodes, both running transactions and spawning new threads for each node. As such, progression through solely local testing was limited.

Fortunately, the Zilliqa development team provided us their testing framework, allowing us to spin up separate nodes in Amazon Web Services. By having separate machines running, we truly tested on a mock of an actual Zilliqa network. This allowed each node to have their own resources, removing any competition for processing time.

Other issues arose from the complexity of understanding, in detail, the relevant parts that need to be changed, as well as the entire system. Adding concurrency required knowledge on creating local network sockets, understanding IPC through JSON-RPC, and understanding the points of transaction failure. The scope of changes extend beyond just modifying Zilliqa. It requires modifying how Scilla servers handle messages and interact with Zilliqa.

6.3 Future Work

There are lots of improvements beyond the proposed fixes in Chapter 5. The immediate next step is to make those changes and ensure that Zilliqa can steadily run transactions concurrently. Briefly, here is a list of steps to be taken to enable concurrent smart contract processing:

- Persistent thread pool at node startup to minimize overhead.
- Allow the Zilliqa to Scilla IPC to handle multiple contracts at once, tying each message to a transaction ID.
- Reconfigure Zilliqa to startup multiple instances of the Scilla server, one for each transaction.

- Ensure that gas accounting is update atomically for shared access between threads.

Next, a new improvement would be adding support for transactions with commutative effects. This entails a much more complicated smart contract state reversion protocol, as well as a more advanced transaction batch partitioning algorithm. Both would heavily rely on CoSplit static analysis.

One possible avenue would be exploring adding concurrency to the Ethereum network. A static analysis for smart contracts in Solidity could provide similar sharding techniques for Ethereum 2.0. From there, the same static analysis can be used for determining safe parallelizable transactions on the local node.

6.4 Conclusion

In this report, we discussed the scalability challenges blockchain protocols face and the motivations of supporting locally-parallel processing of smart contract transactions. We proposed a potential solution for integrating concurrency that relies on disjoint memory footprints of transactions and commutative effects of smart contract transitions through static analysis. By providing some insight and initial implementation on incorporating concurrency in Zilliqa, we have taken a step towards a reliable concurrent transaction processing.

Bibliography

[1] Shehar Bano et al. “Consensus in the age of blockchains”. In: *arXiv preprint arXiv:1711.03936* (2017).

[2] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. “A true concurrent model of smart contracts executions”. In: *CoRR* abs/1905.04366 (2019).

[3] Hung Dang et al. “Towards scaling blockchain systems via sharding”. In: *Proceedings of the 2019 international conference on management of data*. 2019, pp. 123–140.

[4] Thomas D. Dickerson et al. “Adding Concurrency to Smart Contracts”. In: *CoRR* abs/1702.04467 (2017).

[5] Ethereum. *ERC-20 Token Standard*. ethereum.org, Dec. 2020. (Visited on 03/25/2021).

[6] Ethereum. *Ethereum 2.0 (Eth2) vision*. ethereum.org, 2020.

[7] Amrit Kumar, George Pîrlea, and Ilya Sergey. “Practical Smart Contract Sharding with Ownership and Commutativity Analysis”. In: *PLDI* (2021).

[8] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot, 2019.

[9] George Pîrlea. *Ethereum smart contract usage in 7 graphs - George Pîrlea*. pirlea.net, Jan. 2020. (Visited on 03/25/2021).

[10] Vikram Saraph and Maurice Herlihy. "An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts". In: *CoRR* abs/1901.01376 (2019).

[11] *Scalability*. Bitcoin Wiki, 2020. (Visited on 11/12/2020).

[12] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Scilla: a Smart Contract Intermediate-Level LAnguage". In: *CoRR* abs/1801.00687 (2018).

[13] Ilya Sergey et al. "Safer Smart Contract Programming with Scilla". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019).

[14] Nick Szabo. "Formalizing and securing relationships on public networks". In: *First Monday* (1997).

[15] Zilliqa Team. *The Zilliqa Technical Whitepaper*. 2017. (Visited on 11/12/2020).

Appendix A

Crowdfunding Smart Contract

```
scilla_version 0

(* Omitted library functions *)

contract Crowdfunding
(owner: ByStr20, max_block: BNum, goal: Uint128)

field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128

transition Donate ()
  blk <- & BLOCKNUMBER;
  in_time = blk_leq blk max_block;
  match in_time with
  | True  =>
    bs  <- backers;
    res = check_update bs _sender _amount;
    match res with
    | None =>
      msg  = {_tag : ""; _recipient : _sender;
                _amount : Uint128 0; code : already_backed_code};
      msgs = one_msg msg;
      send msgs
    | Some bs1 =>
      backers := bs1;
      accept;
      msg  = {_tag : ""; _recipient : _sender;
                _amount : Uint128 0; code : accepted_code};
      msgs = one_msg msg;
      e = {_eventname : "DonationAccepted";
           donor : _sender; amount : _amount};
      event e;
      send msgs
    end
  | False =>
    msg  = {_tag : ""; _recipient : _sender;
              _amount : Uint128 0; code : missed_deadline_code};
    msgs = one_msg msg;
    send msgs
  end
end
```