RENOIR: Accelerating Blockchain Validation using State Caching

Nitin Awathare
IIT Bombay
nitina@cse.iitb.ac.in

Sourav Das
UIUC
souravd2@illinois.edu

Vinay J. Ribeiro
IIT Bombay
vinayr@iitb.ac.in

Umesh Bellur
IIT Bombay
umesh@cse.iitb.ac.in

ABSTRACT

A Blockchain system such as Ethereum is a peer to peer network where each node works in three phases: *creation, mining,* and *validation* phases. In the creation phase, it executes a subset of locally cached transactions to form a new block. In the mining phase, the node solves a cryptographic puzzle (Proof of Work - PoW) on the block it forms. On receiving a block from another peer, it starts the validation phase, where it executes the transactions in the received block in order to ensure all transactions are valid. This execution also updates the blockchain state, which must be completed before creating the next block. A long block validation time lowers the system's overall throughput and brings the well known Verifier's dilemma into play. Additionally, this leads to wasted mining power utilization (MPU).

Through extensive measurement of 2000 nodes from the production Ethereum network we find that during block validation, nodes *redundantly* execute more than 80% of the transactions in greater than 75% of the blocks they receive - this points to significant potential to save time and computation during block validation.

Motivated by this, we present RENOIR, a novel mechanism that caches state from transaction execution during the block creation phase and reuses it to enable nodes to skip (re)executing these transactions during block validation. Our detailed evaluation of RENOIR on a 50 node testbed mimicking the top 50 Ethereum miners illustrates that when gas limit is increased to 20 times the default value, to accommodate computationally intensive transactions, RENOIR reduces validation time by 90% compared to Ethereum. In addition, throughput of Ethereum reduces from 35326 tx/hour to 24716 tx/hour and MPU from 96% to 67% but these barely change for RENOIR. Furthermore, we deploy a node running RENOIR on the production Ethereum network. Our measurement illustrates that RENOIR reduces the block validation time by as much as 50%.

CCS CONCEPTS

• **Computer systems organization** → **Peer-to-peer architectures**; • **Networks** → *Network measurement*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '21, April 19–23, 2021, Virtual Event, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/3427921.3450247

KEYWORDS

Blockchain; Ethereum; Scalability

ACM Reference Format:
Nitin Awathare, Sourav Das, Vinay J. Ribeiro, and Umesh Bellur. 2021. RENOIR: Accelerating Blockchain Validation using State Caching. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427921.3450247

1 INTRODUCTION

A blockchain system consists of a peer-to-peer network of nodes that agree upon and maintain a state machine with the help of a data structure called a *blockchain*. Each block contains a hash pointer to the previous one, except for the starting "genesis" block, resulting in a chain-like structure of blocks. Each block contains an ordered set of transactions, that when executed updates the state machine [28, 34]. Transactions are generated over time by users, after which they are broadcast in the blockchain network.

Among all outstanding transactions (those not yet included in the blockchain) maintained in a node's transaction pool, it picks a subset and orders them in building a new block. The node also executes these transactions to ensure that its block contains only valid transactions. This is termed the block *creation phase*. Based on the consensus protocol, a block from one of the nodes is chosen as the next one in the chain. This "winning" block is broadcast by the node that created it, and all other nodes execute this block's transactions to ensure that it only contains valid transactions and to update state. This is termed the *validation phase*. The entire process of creating the next block then repeats. In this paper, we primarily focus on PoW (where node solves a cryptographic puzzle) based blockchains but RENOIR can complement other committee based BFT protocol, as explained in §8.

Executing transactions takes a non-trivial amount of time - hence a lengthy and computationally intensive validation phase leads to several performance issues: [8, 21, 26, 34, 36]. **(i)** It slows the process of extending the chain with newer blocks and hence limits the overall throughput of the system. **(ii)** More importantly, high validation time introduces the well known verifier's dilemma [1] in the Proof-of-Work (PoW) based Nakamoto consensus protocols [27] and hence prevents such systems from accepting transactions that require high validation (i.e. computation) time. Also, [5, 14] illustrates that the miners would benefit from not verifying blocks. **(iii)**

[1] Briefly, this refers to a situation in which nodes skip validation of a received block in order to quickly get to the mining phase of the new block they are creating. Doing so risks mining a block on top of a potentially invalid block - hence the dilemma

Lastly, a high validation time also lowers the mining power utilization (MPU) (computation spent on the winning blocks chosen by the consensus protocol) of the overall system resulting in wasted computation. As we show in Section 6, the decrease is more pronounced for nodes controlling a lower fraction of mining power and having weaker connectivity.

In this paper we propose Renoir, an approach to reduce the time required for the validation phase. Renoir is inspired by the following hypothesis: Often, nodes execute the same transactions both during the creation and validation phases. This results in a large number of redundant transaction executions during the block validation phase. Hence, the core idea is to allow nodes to cache the execution result of transactions during the block creation phase, and use the same to avoid (wherever possible) re-execution of transactions during the block validation phase.

Deciding whether it is valid to skip re-executing a transaction is non-trivial. The subtlety primarily arises because the order of transactions in the creation and validation phases can be different and executing transactions in different orders can result in different final states. To address this, each node caches the keys (state variables) and the corresponding values accessed by transactions during the creation phase and uses them during the validation phase in the following manner. For every transaction `tx` in the block being validated, the node checks if it was executed in the creation phase and if the values of the variables read during the execution of `tx` are identical. If so, it skips executing `tx` and updates the state using cached data. In all other cases, the node executes `tx`.

Clearly, the reduction in re-execution of transactions depends on the ordered set of transactions that various nodes pick during the block creation phase. What the scope of this reduction is in production blockchains has so far been an open question. As *our first contribution*, we present extensive measurements of the Ethereum blockchain to answer this question. We pick Ethereum as it is permissionless thereby allowing us to deploy our own measurement nodes, it is popular with more than 7000 active nodes spread across the globe, and supports generic smart contracts. Smart contracts consists of state variables and bundles of instructions, called functions, that a transaction can execute to update the state (§2.2). We estimate the fraction of transactions which 2000 nodes in Ethereum can skip executing in the validation phase for approximately 83 thousand blocks.

Our measurement reveals a high degree of intersection in the transaction sets and their ordering that any node executes during its block creation and validation phases. In particular, we estimate that on average for more than 75% of the blocks, nodes can skip executing at least 80% of transactions during the block validation phase. Our measurements suggest that this is likely due to (i) the low broadcast latency of transactions and (ii) the use of the same default algorithm (that maximizes the block transaction fee) by nodes to select and order transactions in a block.

As a *second contribution*, we design and implement Renoir. As part of Renoir, we propose a novel algorithm to decide when to reuse existing transaction execution results, and prove the correctness of our algorithm. To test the benefits of Renoir, we connect a Renoir equipped Ethereum node to the Ethereum mainnet and determine that the validation time reduces by 50%.

As our *third contribution*, we perform a comprehensive evaluation of Renoir with varying parameters on a network of 50 virtual machines on the Oracle cloud which mimic the 50 top miners of production Ethereum networks that contributes to 99.98% of the total mining power. The goal of these experiments is to study the improvements that Renoir gives with blocks which include computationally intensive transactions, i.e. when Ethereum raises the current gas limit. Our results show that an increase in the gas limit to 20 times the default value raises the validation time to 5 sec while the nodes running Renoir validate the same in a tiny fraction of sec. Results also show that increasing the validation time reduces the throughput of Ethereum from 35326 tx/hour to 24716 tx/hour and MPU from 96% to 67%, but that these remain unaffected for Renoir.

Paper organization. The remaining part of the paper is organized as follows. We describe the necessary background in §2 followed by our approach and findings of Ethereum measurement in §3. We give an overview of Renoir and describe the implementation details in §4. We then theoretically prove the correctness of our design in §5. §6 describes our prototype implementation, experimental setup and evaluation result. We then describe the relevant related works in §7 and conclude with a discussion and further scope for future research in §8.

2 BACKGROUND

In this section, first, we summarize the consensus protocols nodes use to maintain a consistent blockchain. We then describe smart contracts and their execution model. Lastly, we describe the details of block creation, block validation, and propagation of transactions and blocks in Ethereum as they are relevant to the Ethereum measurement study we present.

2.1 The Consensus Protocol

Nodes in a blockchain system form a connected peer-to-peer (P2P) network and run a consensus protocol to append newer blocks to the blockchain. Typically, the consensus protocol is either the Proof-of-Work (PoW) based Nakamoto consensus [28] or the traditional committee based Byzantine Fault Tolerant (BFT) consensus protocol [12, 35]. In this paper we primarily focus on PoW based blockchains, so we provide a brief background on them next. We discuss on how to apply our ideas to committee based BFT protocol in §8.

In PoW blockchains such as Bitcoin [28] and Ethereum [11] each node maintains the same blockchain. Nodes are required to solve a cryptographic puzzle in a process termed *mining*, to append a block to the blockchain. Each node independently mines a block on top of the last block in the longest chain at that node. The resulting mining rate of a node follows a Poisson process with a rate proportional to the mining power controlled by the node [19]. The length of the longest chain at a node is termed the height of the chain. Occasionally, the chain at a node diverges into two paths, a process termed *forking* and can occur as a result of two or more miners mining a block at the same time. In the presence of multiple paths, a new block is mined on top of the longest path.

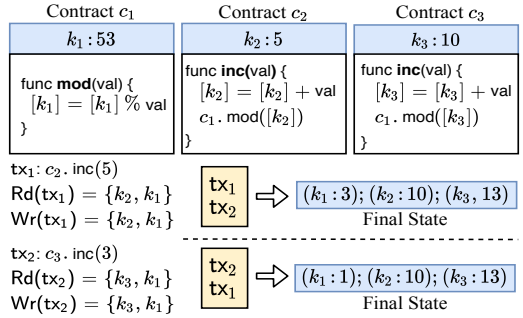


Figure 1: Contract c_1, c_2 and c_3 with their initial state $\{(k_1 : 53), (k_2 : 5)$ and $k_3 : 10)\}$, respectively. Transaction tx_1 and tx_2 are two transactions that update these states according to the specified program logic.

2.2 Smart Contracts & Transactions

Smart contracts are program logic structured as functions. Each smart contract has a unique cryptographic *address*. A smart contract is created when a transaction containing its description is included in the blockchain. Once deployed, the functions of the contract can be invoked by any transaction by referring to the address of the contract, the identity of the function, and supplying its parameters. A function in a smart contract can call a function in another smart contract. Each smart contract maintains state, a disjoint set of key-value pairs. The state maintained by a smart contract is read and modified when transactions that address the contract are executed. For every transaction tx, the set of keys read and written by tx during its execution is termed the *read-set* and *write-set* of tx respectively. We refer to the read-set and write-set of a transaction tx as Rd(tx) and Wr(tx), respectively.

Figure 1 illustrates three smart contracts c_1, c_2, and c_3 along with their state $(k_1 : 53), (k_2 : 5)$, and $(k_3 : 10)$, respectively. Here on, for any key k, we will use $[k]$ to indicate the value stored at the key. For example, $[k_1]$ is 53. Contract c_2 and c_3 have a function named inc that takes in an integer val as its input and adds val to the content of their respective keys. Contract c_1 has a function mod which takes in a integer input val and update the contents of its key k_1 with $[k_1]$%val. Consider two transactions tx_1 and tx_2 as described in Figure 1. These transactions can be included in the blockchain in any chosen order. Indeed, if they were included in the order tx_1 followed by tx_2, then the resulting state of the system will be $(k_1 : 3), (k_2 : 10)$, and $(k_3 : 13)$. On the other hand, if the transactions were included in the order tx_2 followed by tx_1, the resulting state of the system will be $(k_1 : 1), (k_2 : 10)$, and $(k_3 : 13)$. However, in both possible chosen order, the read-set and write-set of the transactions are same.

2.3 Block Creation and Validation in Ethereum

A transactions is created at a node which then broadcasts it. Upon receiving new transactions, nodes validate and adds them to their *transaction pools*.

Block creation. To create a block at height i, a node r picks an ordered list of transactions $\mathbf{T}'$ from its transaction pool by prioritizing the transactions paying higher fees. Next, r executes all the transactions from $\mathbf{T}'$ in the order they appear in $\mathbf{T}'$. This execution modifies the state of any smart contracts that the transactions operate on. Finally, r starts mining on the block containing these transactions. If r successfully solves the cryptographic puzzle for the block at height i before receiving any other valid block at the same height, r broadcasts its block and proceed to create the next block at height $i + 1$. As transactions continue to arrive at r during mining, r periodically updates $\mathbf{T}'$ to include a newly arrived transaction that offers higher fees and restarts the mining process, if not complete. The period after which r updates the ordered list is referred to as the *recommit interval*.

Block validation. During the mining process, if r receives a new block, say B_i, at height i that includes the ordered list of transactions $\mathbf{T}$, r abandons its mining process and validates the received block as follows. First, r validates that the creator of B_i successfully solved the cryptographic puzzle. Then, r executes the transactions included in $\mathbf{T}$ in the order they are placed in the block. On successful validation, r discards the execution result of $\mathbf{T}'$ and proceeds to create the next block at height $i + 1$ on top of B_i. Validation can be computationally intensive and the time taken to execute the transactions during validation is termed the *block validation time*.

Serial nature of validation, creation and consensus: On receiving a block, say B_i, at height i from other nodes in the network, the node r validates B_i. Next, r creates the potential block B'_{i+1} at a height $i + 1$. Finally, r starts PoW(mining) on the B'_{i+1}.

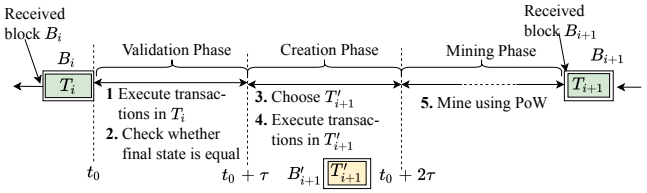


Figure 2: Serialization of the block validation phase, block creation phase and mining phase

.

As the transactions included in B_i update the state, an attempt to create the subsequent potential block B'_{i+1} before the validation of B_i, may result in an inconsistent state. This makes the validation of the received block and creation of a new block on it a serial process. For example as illustrated in Figure 2, r starts validating B_i at time instance t and finishes it at $t + \tau$, where τ is the block validation time. Assuming block validation and creation takes equal amount of time, creation of block B'_{i+1} ends at $t + 2\tau$. Lastly, mining starts on B'_{i+1}.

Increase in block validation time delays the creation of a subsequent potential block further and hence PoW on it. This reduces the time a node can invest for PoW and accordingly reduces the chance of solving the cryptographic puzzle, more so for a node with a lesser mining power.

2.4 Transaction/Block Propagation in Ethereum

Nodes in Ethereum run one of the available implementations of Ethereum protocol [3]. We observe that the protocols followed by

nodes vary slightly with different implementations. Nevertheless, in this paper we will focus on only the go-ethereum [4] and parity-ethereum [31] implementations as more than 93% of the nodes in Ethereum run one of these two implementations.

In these implementations, on receiving a new *transaction* from another node, the receiver node first validates the transaction by checking that it is appropriately signed, adheres to the required encoding and pays the appropriate fees. Upon successful validation, the node adds the transaction to its transaction pool and immediately broadcasts it to its peers. Nodes running go-ethereum broadcast it to all their adjacent peers while parity-ethereum nodes broadcast it to a randomly chosen square root number of its adjacent peers.

Both go-ethereum and parity-ethereum nodes follow the same protocol to broadcast received *blocks*. In particular, a node r with N adjacent peers, first broadcasts every received block to a randomly chosen $\sqrt{N}$ of its peers. Next, r validates the transactions included in the block by executing them. On successful validation, r broadcasts a hash of the block to all the remaining peers.

This propagation model makes it highly likely that transactions pools from which new blocks are created at different nodes are similar - a property that we can exploit in avoiding redundant transactions.

3 EMPIRICAL STUDY OF ETHEREUM

In this section, we will first describe our approach to measure the potential benefits of avoiding re-execution of transactions in Ethereum network. Then, we will present our measurement results and analyze them in detail.

3.1 Definitions

Let Rd(tx) be the read set of a transaction tx computed during its execution in the block creation phase. Recall that the read set of a transaction consists of keys read by the transaction during its execution. During validation of a received block B a node r skips re-execution of transaction tx only if the values stored in keys from Rd(tx) has not been modified by any preceding transaction in B. Hereon, we refer to such transactions as a *unaltered transaction*.

In Ethereum, the time taken to execute every transaction can be different and is measured in *gas*, a unit of computation. Every transaction has a parameter *used gas* which is proportional to the amount of time needed to execute the transaction. Since our goal is to measure the reduction in time required to validate a block, which in turn depends on the gas used by skipped transactions, we define the notion of *similarity* of a block. Formally,

Definition 3.1. *(Similarity)* For a received block B with ordered list of transaction $\mathbf{T}$, its similarity is the ratio of total gas used by unaltered transactions in $\mathbf{T}$ to total gas used by all the transactions in $\mathbf{T}$. Let $\mathbf{U} \subseteq \mathbf{T}$ be the subset of unaltered transactions in $\mathbf{T}$, then,

$$\text{Similarity of block } B = \frac{\sum_{\forall \texttt{tx} \in \mathbf{U}} \text{Gas used by tx}}{\sum_{\forall \texttt{tx} \in \mathbf{T}} \text{Gas used by tx}} \quad (1)$$

3.2 Measuring Similarity

Observe that to compute similarity of a Ethereum block at any given node r, one needs to know the (i) set of transaction r executes during the block creation phase, and; (ii) the order in which r executes these transactions. Since existing Ethereum nodes, here on referred to as the *host nodes*, do not reveal the set of transactions they execute during the block creation phase to any of its peers, measuring similarity at host nodes turns out to be non-trivial.

We address this challenge by developing a heuristic to approximate the similarity at host nodes. Our heuristic exploits the fact that host nodes in Ethereum immediately broadcast the transactions and blocks they receive to a subset of its peers (§2.4). Recall from §2.3, Ethereum host nodes add new transactions to their transaction pool from which they pick a subset of transactions ordered by the fee each transaction pays. This suggests that if we can estimate the transaction pool of a host node r, we can use its contents to identify the ordered list of transactions r would execute during the block creation phase. Indeed we follow this approach in our measurement.

We connected 10 new instrumented Ethereum nodes, here on referred to as the *probe nodes* and denoted using $\{n_1, \cdots, n_{10}\}$, to approximately 2000 distinct host nodes spread across different geographical locations on the Ethereum mainnet. Each probe node ran on a Oracle virtual machine (VM) with 16 2.1 GHz cores, 120GB memory, 6.4TB NVMe SSD, 8.2 Gbps network bandwidth running Ubuntu 16.04 and go-ethereum version 1.9.0. Each probe node was deployed in one of the following ten different geographical regions: US East (Ashburn), US West (Phoenix), Switzerland (Zurich), UK (London), Australia (Sydney), Japan (Tokyo), Canada (Toronto), India (Mumbai), Brazil (Sao Paulo), and Germany (Frankfurt).

Transaction pool estimation. For every host node r that was connected to one of our probe nodes, say n_1, we measured the similarity at r as follows. First, based on the IP address of r, we estimated the geographical location of r and used this location to determine the network latency, here on referred to as latency, between r and n_1. We determined latency using [1]. Also, from our measurements, we found that on average a host node takes approximately 40 milliseconds to validate a block header and approximately 200 milliseconds to validate an entire block. Here on, we refer to them as headValidTime and blkValidTime, respectively.

For every transaction n_1 receives from host node r at time t_{n_1}, n_1 estimates the time instant, t_r, when r received the same transaction to be $t_r = t_{n_1} - \text{latency}$. Similarly, when n_1 receives information about a new block from r at time t_{n_1}, we estimate t_r to be either $t_r = t_{n_1} - \text{latency} - \text{headValidTime}$ or $t_r = t_{n_1} - \text{latency} - \text{blkValidTime}$ depending upon whether we receive the entire block or only the hash of the block. We used the same value of latency for both block and transaction because, t_{n_1} in our case is the arrival time of first network packet from r. Note that, the latency of first packet in any communication channel is independent of the total size of the message being transmitted.

We then used our estimation of t_r for the received transactions and blocks to estimate the transaction pool of host nodes. In particular, at any given time instant t, the transaction pool at node r consists of all the transactions that arrive at r before t and are not included in any of the blocks. For example, in Figure 3, according to our estimate, by time t_{c_1}, r receives the transaction $\texttt{tx}_0$ and $\texttt{tx}_1$. Since r also receives and validates the block B' which include $\texttt{tx}_0$ by time t_{c_1}, the transaction pool of r at time t_{c_1} contains only $\texttt{tx}_1$. Similarly, transaction pool of r at time t_{c_2} contains $\texttt{tx}_1$ and $\texttt{tx}_2$.

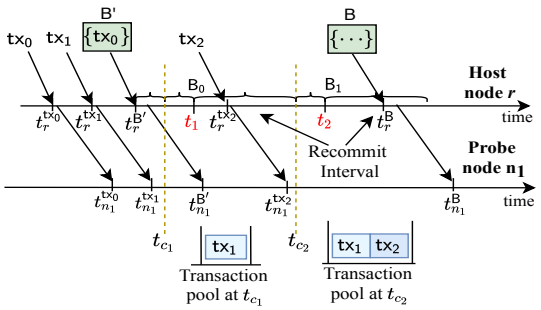


Figure 3: Transaction and block flow between host node and probe node

Computing similarity. We used our estimated transaction pool to pick the ordered list of transactions a host node would have executed during the block creation phase and use that to compute the similarity of every received block. In our estimation, we also accommodate the fact that nodes in Ethereum repeatedly execute a new set of transactions after every recommit interval (ref. §2.3). In particular, for a node r we measure the similarity of a received block B at r with the ordered list of transactions chosen at time instant t. Here, t is the latest time instant when r executed the new ordered list transactions from its transaction pool. t is measured by incrementing the time in multiples of ~3 seconds (3 seconds+block creation time) from the validation of parent block of B. Specifically, let B' be the parent block of B, and blkCreateTime is the time required to create a block. Also, let $t_r^{B'}$ and t_r^{B} be the time of arrival of block B' and B, respectively. Then we pick $t = t_r^{B'}$ +blkValidTime+ $k\times$(3+blkCreateTime) for the largest value of k that satisfies $t < t_r^{B}$.

For example, in Figure 3, after validating block B', r will create a new block B_0 and start mining. When the recommit interval expires at time t_{c_2}, say r updates its block to B_1 which includes the newly arrived transaction $\texttt{tx}_2$. Upon arrival of the block B at r, we compute the similarity of B with B_1, the block r creates during its latest recommit.

Validating the approach. To validate the approach we deployed a 11^{th} probe node, say n_{11}. Then, we treated the first ten probe nodes, $\{n_1, \cdots, n_{10}\}$ as host nodes and estimated the similarity for these ten nodes using n_{11}. Next, we computed the actual similarity at n_1 to n_{10} and compared it against the similarity we estimated using n_{11}.

We report our detailed findings and analysis from our measurement in the next section.

3.3 Findings & Analysis

Our measurements from October 3 to October 9, 2019, using our probe nodes recorded data for a duration of approximately 83 thousand Ethereum block intervals. The data consists of blocks and transactions along with the estimated time instant they arrive at each neighboring host node.

Figure 4 illustrates our estimated similarity at host nodes running go-ethereum. We observe a similarity of more than 80% for more than 75% of the blocks at go-ethereum hosts. However, estimation of similarity at hosts running parity-ethereum is not possible as

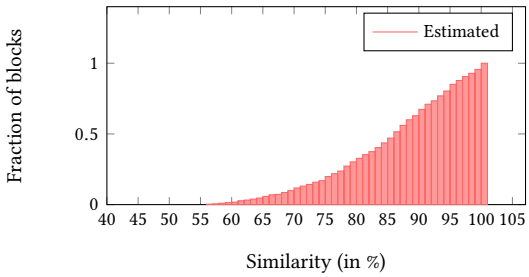


Figure 4: Cumulative average estimated similarity at host nodes running go-ethereum **implementation.**

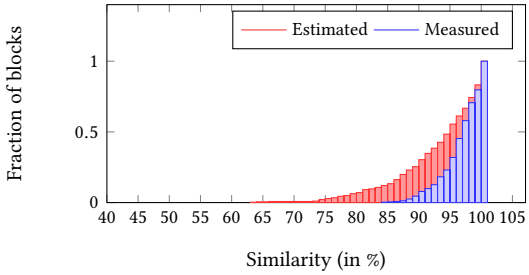


Figure 5: Cumulative similarity at our instrumented go-ethereum **nodes $\{n_1, \cdots, n_{10}\}$. The measured similarity reports the actual similarity observed at these nodes and the estimated similarity reports our estimation of similarity at n_1 to n_{10} using a different probe node n_{11}.**

hosts running parity-ethereum only broadcast transactions to a square root number of its total peers (ref §2.4). Thus here on, we will only illustrate our measurements for go-ethereum hosts.

Figure 5 illustrates findings that validate our process for estimating similarity. In particular, we measure the true similarity at nodes n_1 to n_{10} and compare it against the similarity estimated at these nodes by our additional probe node n_{11}. Note that, in our measured scenario, we observe more than 90% similarity for almost all blocks. Furthermore, we observe that the estimated similarity is significantly lower than the true similarity. Upon close inspection, we find that n_1 to n_{10} were dropping approximately 10% of the total transactions before sending them to our probe node n_{11}. This is because host nodes uses a queue of size 128 for each of its peers to send transactions to them and drops transactions whenever the queue is full. This suggests that true similarity at Ethereum host nodes are likely to be significantly greater than our estimated values.

Recall from §3.2, that Ethereum hosts periodically recommits the ordered list of transactions they process to include newly arrived high fee transaction. Because the block propagation takes around 3 seconds to reach 95% of the network (Figure 8), host nodes may compute the similarity of the received block with the chosen order list of transaction at different recommits. Hence, a natural question is how does the similarity vary across different recommits. Furthermore, it also raises concerns about the incorrect estimation of time to recommit in our measurements. To address these questions, we

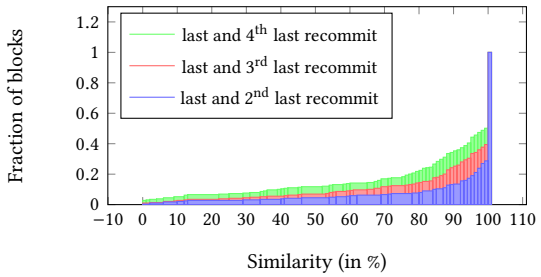


Figure 6: Cumulative similarity between ordered list of transactions chosen for measuring the similarity at a host node and the ordered list chosen at different recommits. Results are averaged over all hosts nodes running go-ethereum **implementation.**

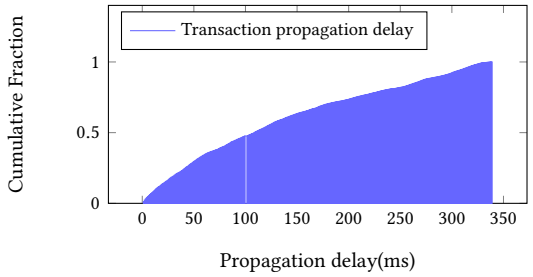


Figure 7: Transaction propagation delay in Ethereum network.

estimate the similarity between the ordered list of transaction we pick to compute similarity of host nodes, i.e., the *last* recommit, with ordered list of transactions chosen at previous recommit instances. We identify the time instants of recommits based on our approach described in §3.2. Figure 6, illustrates the average similarity between ordered list of transactions at last recommits and three previous recommits. Observe that the value of similarity does not change much across last four recommits.

Other findings. As a part of our case study, we also compute the propagation delay of transactions and blocks in Ethereum, which might be of independent interest to the reader. We measure the propagation delay of a transaction as the elapsed time between the first time one of our probe nodes hears about the transaction and the time our probe nodes hear the same transaction from 95% of the Ethereum node they are connected with. We use the same approach to compute the block propagation delay as well. Figure 7 and Figure 8 presents the measured transaction and block propagation delay, respectively.

4 RENOIR DESIGN

The core idea of Renoir is to cache the results of executing transactions during the block creation phase and use the same to eliminate (re)executing the same transactions during the block validation phase. The challenge arises from the fact that transactions often depend on each other. As a result, a difference in the order of transactions in the received block and those executed during the creation

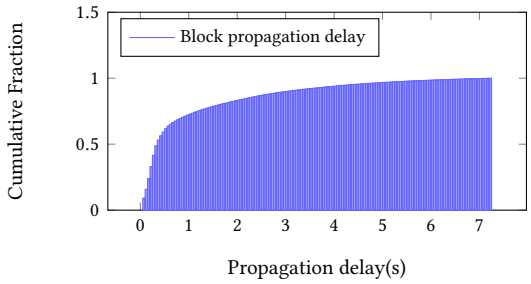


Figure 8: Block propagation delay in Ethereum network.

phase can result in different final states. Renoir must also ensure that it does not miss executing new transactions that the node has not executed during the block creation phase.

4.1 Overview

Renoir addresses these issues in the following manner. Recall from §2.3 that for every new block at height i, during its block creation phase the node picks a new ordered list of transactions $\mathbf{T}'$ and executes them. For each transaction in the ordered list, the node computes its read and write sets during its execution and caches it locally. Further, for every key in the read set of each transaction, the node maintains the order of transactions that writes to the key.

Upon receiving a new block at height i containing an ordered list of transactions $\mathbf{T}$, a node in Renoir uses the cached information to process transactions in $\mathbf{T}$. In particular, for each transaction, `tx`, in the received block, if r has not executed `tx` during the block creation phase, i.e., $\texttt{tx} \notin \mathbf{T}'$, r executes `tx`. Otherwise, r skips executing `tx` *if* the transactions preceding `tx` in the received block accesses (writes to the values corresponding to) the keys in the read-set of `tx` in same order as they did during the block creation phase. The idea is, if the order of transactions that accesses the keys in the read set of a transaction are identical during both the block creation and validation phase, the execution result of the transaction remains reusable during the block validation phase.

In our implementation, we use two tables, namely, *transaction* table and *key* table to maintain the desired information. The transaction table maintains the read and write set of every transaction executed during the block creation phase. Similarly, the key table maintains the order of transactions that accesses the keys in read and write sets of each transaction. Further, the key table also maintains the values written by the transactions during the block creation phase. During block validation phase, nodes use these two tables to skip only those transactions whose access order are identical to the access order during the block creation phase. Also, once a node decides to skip a transaction, it uses the values stored in the key table to update the state touched by that transaction. The updated state is used by future transactions. We prove that Renoir ensures the correct final state in §5. Renoir maintains these table for each block till its gets confirmed, i.e., appended by six blocks. So Renoir node at any point of time contains six such pairs of tables.

Next we will describe the data-structures of the transaction table and key table along with the algorithms a Renoir node follows

during block creation phase and block validation phase. Finally, we illustrate the protocol with an example.

4.2 Data Structures

Here on we refer to the transaction and key table with TxTable and KeyTable, respectively.

Transaction table. TxTable stores a row for each transaction executed during the block creation phase. Each row contains a list of triples, L_{trans}, comprising one triplet for every key accessed by the transaction during its execution. In particular, for a transaction tx and for a key k accessed by tx, the corresponding triple stores $\langle k, \text{ver}(k), \text{isWrite}\rangle$, where ver($k$) denotes the version of k accessed by tx, and isWrite is a bit that records whether the tx writes to k or not. By default, ver(k) of every key is initialized to 0. We use ver(k) and isWrite for every key to maintain the order in which the key is accessed during the block creation phase. We then use this order to decide whether to skip re-executing a transaction. Note that, a naive approach of comparing values at keys to take this decision would not work as values stored at any key could be arbitrarily large.

Key table. KeyTable stores a separate row for each key accessed by the transactions during the block creation phase. For a key k, the corresponding row contains a list of pairs $\langle \text{ver}(k), v\rangle$ where ver(k) is the version of the k accessed by tx and v is the value of k written by tx. We refer to this list of pairs as L_{lis}. If tx does not overwrite the value associated with k, we store $\perp$ as the value of v. Each row also stores the index and isValid, and they are initialized with 0 and True, respectively.

Algorithm 1 TableCreation

```
1:  input T' = {tx'_1, tx'_2, ..., tx'_m'}
2:  TxTable ← {}, KeyTable ← {}
3:  for each transaction tx in T' do
4:      Wr, Rd← execute(tx')
5:      UpdateTables(tx', Wr, Rd)
6:  return (TxTable, KeyTable)
7:
8:  procedure UpdateTables(tx, Wr, Rd)
9:      add tx to TxTable
10:     for each key k in Rd ∪ Wr do
11:         if k not in KeyTable then
12:             add k to KeyTable
13:         if k in Wr then
14:             add ⟨k, ver(k), 1⟩ to L_trans of tx
15:             add ⟨ver(k), Wr(k)⟩ to L_lis of k
16:             ver(k) ← ver(k) + 1
17:         else
18:             add ⟨k, ver(k), 0⟩ to L_trans of tx
19:             add ⟨ver(k), ⊥⟩ L_lis of k
```

4.3 Block Creation Phase

Let r be the node and $\mathbf{T}'$ be the ordered list of transaction chosen by r to create a new block. For each transaction tx $\in \mathbf{T}'$, r adds a new row in the TxTable. While executing tx, r identifies the keys accessed by tx, and for every key k, it adds a tuple $\langle k, \text{ver}(k), \text{isWrite}\rangle$. The ver($k$) is the version of the key k (and the corresponding value) before executing tx and starts with 0. Similarly, isWrite is set to True if tx writes to the value associated with the k during its execution. After executing tx, r increments the version numbers of all the keys whose values are overwritten by tx.

For those keys accessed by tx during the block creation phase, r updates KeyTable as follows. For each key that is present in the KeyTable, r append the pair $\langle \text{ver}(k), v\rangle$ to its L_{lis}. On the other hand for each key that is not already present in the KeyTable, r adds a new row containing the tuple $\langle \text{ver}(k), v\rangle$, where ver($k$) is the version of k before executing tx and v is the value tx writes for k. r stores $\perp$ in k if tx does not overwrite the value associated with the k. Procedure UpdateTables in Algorithm 1 describes the pseudo-code updating the TxTable and KeyTable.

4.4 Block Validation Phase

Upon receiving a new block B containing an ordered list of transactions $\mathbf{T}$, r processes the transactions in $\mathbf{T}$ in the order they appear. For a transaction tx, if r has not executed it during the block creation phase, i.e., tx $\notin \mathbf{T}'$, r executes tx and computes its read write set. Let Wr(tx) be the corresponding write set. Then, for every key k in Wr(tx), r invalidates the row corresponding to k in KeyTable by setting the value of isValid variable to be False.

If the transaction tx is present in both $\mathbf{T}$ and $\mathbf{T}'$, r skips tx only if the sequence of transactions that modifies the values associated with the keys in Rd(tx) is identical in both $\mathbf{T}$ and $\mathbf{T}'$. In particular, while processing the transaction tx, r checks that for every key k in the L_{trans} of tx, the corresponding row in KeyTable is still valid, and the current version of these keys matches the version stored in L_{trans}. We use the index stored in the row corresponding to KeyTable to identify the current version of the key. If both conditions are satisfied then r skips executing tx, and increments index for all the keys in read-set of tx. Also, r uses the value v stored in the L_{lis} to update the state appropriately. If any of these conditions is not satisfied, then r invalidates the rows corresponding to keys in write-set of tx. Also, r re-executes tx, computes its new write set, and invalidates rows for all the keys in the new write set. Algorithm 2 illustrates the pseudo-code of validating a received block with ordered list of transactions $\mathbf{T}$.

4.5 An Example

Let $\mathbf{T}' = \{\text{tx}_1, \text{tx}_2, \text{tx}_3\}$ be the ordered list of transactions that a node r executes during the block creation phase. Tables 1(a), 1(b), and 1(c) illustrate the read-write set information for transactions in $\mathbf{T}'$, the TxTable, and the KeyTable created using transactions in $\mathbf{T}'$ respectively. Since, tx_1 is the first transaction in $\mathbf{T}'$, version of all the keys in its access list is zero. Moreover, since tx_1 writes to address k_1 and k_2, the version of key k_1 in tx_2 is 1. Also, since no transaction writes to key k_3, the isWrite for k_3 is zero everywhere.

Upon receiving a block with ordered list of transactions $\mathbf{T} = \{\text{tx}_1, \text{tx}_4, \text{tx}_3\}$ with read and write sets as shown in Table 2, r processes them as follows. Since tx_1 is present in both $\mathbf{T}$ and $\mathbf{T}'$, its read-write set from the block creation phase is stored in TxTable. Furthermore, the row corresponding to each key in the read set of tx_1 is valid in KeyTable. Also, the version of each key in the

Index	Transaction	Read set	Write set
1	tx_1	$\{k_1, k_2\}$	$\{k_1 : 3, k_2 : 2\}$
2	tx_2	$\{k_1, k_3\}$	ϕ
3	tx_3	$\{k_1, k_2, k_3\}$	$\{k_2 : 4\}$

(a)

Transaction	$\{\langle$key k, ver(k), isWrite$\rangle\}$
tx_1	$\langle k_1, 0, 1\rangle; \langle k_2, 0, 1\rangle$
tx_2	$\langle k_1, 1, 0\rangle; \langle k_3, 0, 0\rangle$
tx_3	$\langle k_1, 1, 0\rangle; \langle k_2, 1, 1\rangle; \langle k_3, 0, 0\rangle$

(b)

Key k	$\{\langle$ver(k), $v\rangle\}$	index	isValid
k_1	$\langle 0, 3\rangle; \langle 1, \perp\rangle; \langle 1, \perp\rangle$	0	True
k_2	$\langle 0, 2\rangle; \langle 1, 4\rangle$	0	True
k_3	$\langle 0, \perp\rangle; \langle 0, \perp\rangle;$	0	True

(c)

Table 1: (a) Read and write set of transactions tx_1, tx_2 and tx_3 from T′, when they are executed in the order specified by the index. (b) TxTable **for transactions in T′, and (c)** KeyTable **for transactions in T′.**

Algorithm 2 BlockValidation

```
1:  input T = {tx_1, tx_2, ..., tx_m}
2:  for each transaction tx in T do
3:      if tx in TxTable then
4:          if CanSkip(tx) then
5:              for each key k in write-set of tx do
6:                  commit value of k
7:              continue
8:          for each key k in write-set of tx do
9:              set isValid of k in KeyTable to be False
10:     Wr, Rd ← execute(tx)
11:     for each key k in Wr do
12:         set isValid of k in KeyTable to be False
13:
14: procedure CanSkip(tx)
15:     for each key k in the write-set of tx do
16:         if isValid of k in KeyTable is False then
17:             return False
18:         if ver(k) mismatch in KeyTable & TxTable then
19:             return False
20:     return True
```

Index	Transaction	Read set	Write set
1	tx_1	$\{k_1, k_2\}$	$\{k_1 : 3, k_2 : 2\}$
2	tx_4	$\{k_2\}$	$\{k_2 : 3\}$
3	tx_3	$\{k_3\}$	$\{k_2 : 4\}$

Table 2: Read and write set of transactions tx_1, tx_4 and tx_3 from the ordered list of received block, when their are executed in the order specified by the index.

read-set matches the version in the KeyTable. Hence, r skips re-executing tx_1. As tx_4 is not present in $\mathbf{T}'$, r will execute tx_4 and invalidate all the rows corresponding to the write set of tx_4. In particular, from Table 2, the write-set of tx_4 consists of k_2, hence, r will invalidate the KeyTable row corresponding to k_2. As a result of this invalidation, the first condition for skipping tx_3 gets violated, i.e., rows for all keys in its key-set is no longer valid. Hence, r will re-execute tx_3. Note that at the end of block validation phase the final state at node r is equivalent to the state of re-executing all the transactions in $\mathbf{T}$.

5 PROOF OF CORRECTNESS

In this section, we will prove that the block validation procedure of Renoir produces a state equivalent to the state of naive re-execution of all the transactions in the received block. We will first show that nodes in Renoir skip re-executing a transaction, tx, only when the sequence of transactions writing to each key in the read set of tx is identical with the sequence of transactions that accesses these keys during the block creation phase. We then show that all the transactions writing to the set of keys have been skipped during the block validation phase.

Let state^N_{i-1} and state^R_{i-1} be the resulting states after executing all the transactions till the block at height $i-1$ using naive re-execution and using Renoir validation, respectively. Also, for a block at height i, let $\mathbf{T}'$ and $\mathbf{T}$ respectively, be the ordered list of transactions a node r executes during the block creation phase and the ordered list of transactions included in the received block. We will prove that starting with an identical state at block height $i-1$, i.e., $\text{state}^N_{i-1} = \text{state}^R_{i-1}$, the resulting states after block height i are identical, i.e., $\text{state}^N_i = \text{state}^R_i$.

Definition 5.1. (*Write Sequence*) For an ordered list of transaction $\mathbf{T}$ and a transaction $\texttt{tx} \in \mathbf{T}$, the write sequence of a key $k \in \texttt{Rd(tx)}$ is the ordered list of transactions from $\mathbf{T}$ that appear before tx and write to k during their execution. We use $\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k)$ to denote the corresponding write sequence. We use $|\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k)|$ and $\{\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k)\}$ to denote the number and the unordered set of transactions in $\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k)$.

Lemma 5.1. During the block validation phase, Renoir skips re-executing a transaction $\texttt{tx} \in \mathbf{T} \cap \mathbf{T}'$, only if both $\{\text{WSQ}^{\texttt{tx}}_{\mathbf{T}'}(k)\}$ and $\{\text{WSQ}^{\texttt{tx}}_{\mathbf{T}'}(k)\}$ are identical.

Proof. Renoir skips tx only when version of every key in the Rd(tx) matches in the TxTable and KeyTable. Since, version of a key is incremented only during a write access, this implies that the number of transactions in the write sequence of each key in Rd(tx) are identical in $\mathbf{T}$ and $\mathbf{T}'$, i.e.,

$$|\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k)| = |\text{WSQ}^{\texttt{tx}}_{\mathbf{T}'}(k)|, \forall k \in \texttt{Rd(tx)} \tag{2}$$

For every key $k \in \texttt{Rd(tx)}$, every transactions in $\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k)$ is also present in $\text{WSQ}^{\texttt{tx}}_{\mathbf{T}'}(k)$. Otherwise, for each transaction in $\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k) \setminus \text{WSQ}^{\texttt{tx}}_{\mathbf{T}'}(k)$, Renoir would have invalidated k, and hence would not have skipped tx. This implies that when treated as sets,

$$\text{WSQ}^{\texttt{tx}}_{\mathbf{T}}(k) \subseteq \text{WSQ}^{\texttt{tx}}_{\mathbf{T}'}(k), \forall k \in \texttt{Rd(tx)} \tag{3}$$

Combining equation 2 and 3, we get the desired result. □

Lemma 5.2. During the block validation phase, Renoir skips re-executing a transaction $\texttt{tx} \in \mathbf{T} \cap \mathbf{T}'$ only when all the transactions in the write sequence of every key in Rd(tx) were skipped during the block validation phase.

Proof. For the sake of contradiction, assume that there exists a transaction $\mathtt{tx}' \in \mathrm{WSQ}_{\mathbf{T}}^{\mathtt{tx}}(k)$ for some key $k \in \mathtt{Rd}(\mathtt{tx})$ that has been re-executed during the block validation phase. Then Renoir would have invalidated KeyTable rows for all the keys (including k) from the write set of $\mathtt{tx}'$. This implies that Renoir would not have skipped tx, thus leading to a contradiction. □

Theorem 1. *When* Renoir *skips re-executing a transaction* $\mathtt{tx} \in \mathbf{T} \cap \mathbf{T}'$*, the write sequence for every key in* $\mathtt{Rd}(\mathtt{tx})$*, is identical for both* $\mathbf{T}$ *and* $\mathbf{T}'$*, i.e.,*

$$\mathrm{WSQ}_{\mathbf{T}}^{\mathtt{tx}}(k) = \mathrm{WSQ}_{\mathbf{T}'}^{\mathtt{tx}}(k), \forall k \in \mathtt{Rd}(\mathtt{tx}) \tag{4}$$

Proof. For the sake of contradiction assume that this is not true for some key $k \in \mathtt{Rd}(\mathtt{tx})$. But, from Lemma 5.1, we know that $\mathrm{WSQ}_{\mathbf{T}}^{\mathtt{tx}}(k)$ and $\mathrm{WSQ}_{\mathbf{T}'}^{\mathtt{tx}}(k)$ has same set of transactions. Furthermore, from Lemma 5.2, we know that Renoir skips every transaction in them.

Let $\mathrm{WSQ}_{\mathbf{T}}^{\mathtt{tx}}(k)$ and $\mathrm{WSQ}_{\mathbf{T}'}^{\mathtt{tx}}(k)$ differ for the first time at an index l. Let $\mathtt{tx}' \in \mathrm{WSQ}_{\mathbf{T}'}^{\mathtt{tx}}(k)$ and $\mathtt{tx}'' \in \mathrm{WSQ}_{\mathbf{T}}^{\mathtt{tx}}(k)$ be the corresponding transactions. This implies that in $\mathrm{WSQ}_{\mathbf{T}'}^{\mathtt{tx}}(k)$, $\mathtt{tx}''$ appear later than $\mathtt{tx}'$. Thus, in $\mathrm{WSQ}_{\mathbf{T}'}^{\mathtt{tx}}(k)$, $\mathtt{tx}''$ reads a version of k strictly greater l. Hence, according to the specification, Renoir will detect a version mismatch for $\mathtt{tx}''$ during block validation phase, and hence will re-execute $\mathtt{tx}''$. This contradicts Lemma 5.2. □

Starting from $\mathrm{state}_{i-1}^{N} = \mathrm{state}_{i-1}^{R}$, let $\mathrm{state}_{i-1}^{N}(u)$ and $\mathrm{state}_{i-1}^{R}(u)$ be the state of the transaction after executing the u^{th} transaction in $\mathbf{T}$ using the naive approach and Renoir approach, respectively. Next, we will use Theorem 1 to prove that the block validation in Renoir produces a state equivalent to the state produced by naively re-executing every transaction in the received block.

Theorem 2. *Starting with initial state* $\mathrm{state}_{i-1}^{N} = \mathrm{state}_{i-1}^{R}$*, for every received block at height* i *with ordered list of transactions* $\mathbf{T}$*,* Renoir *and the approach of naive re-execution of all transactions in* $\mathbf{T}$ *will result in identical final state, i.e.,* $\mathrm{state}_{i}^{N} = \mathrm{state}_{i}^{R}$

Proof. It is obvious that until Renoir skips re-executing any transaction in $\mathbf{T}$, the state in Renoir and naive re-execution remains identical. Hence, we start with the first transaction that Renoir skips and then prove the theorem via induction.

Base case. Let u be the index of first transaction Renoir decides to skip and $\mathtt{tx} = \mathbf{T}[u]$. Hence, the write sequence of every key in $\mathtt{Rd}(\mathtt{tx})$ is empty. This implies that the contents of keys in $\mathtt{Rd}(\mathtt{tx})$ have not been modified by any preceding transaction of tx. Thus, execution of tx will result in same state update as in the creation phase. Hence the final state after skipping tx will be identical to naive re-execution.

Induction hypothesis. For any index $v \geq u$, assume that the state after executing transactions up to (including) v have resulted in the correct state, i.e.,

$$\mathrm{state}_{i-1}^{N}(v) = \mathrm{state}_{i-1}^{R}(v) \tag{5}$$

Inductive step. Let $\mathtt{tx}' = \mathbf{T}[v+1]$. If Renoir decides to re-execute $\mathtt{tx}'$, then trivially $\mathrm{state}_{i-1}^{N}(v+1) = \mathrm{state}_{i-1}^{R}(v+1)$. Otherwise, from Theorem 1 we know that the write sequence of for all keys in $\mathtt{Rd}(\mathtt{tx}')$ are identical to the block creation phase. Furthermore, from Lemma 5.2 Renoir skips all transactions that appear in these write sequences. Since the last transaction writing to every key in $\mathtt{Rd}(\mathtt{tx}')$ are the same in both the block creation and the block validation phase, and Renoir skips all of them, the value stored in these keys are also identical in both phases. Stating differently, the contents stored at all keys in $\mathtt{Rd}(\mathtt{tx}')$ are identical. Since tx is deterministic and its execution only depends on the values stored at the keys in its read set, the resulting state will be identical. This proves the theorem. □

6 RENOIR EVALUATION

We implemented Renoir on the open-source Go-Ethereum client version 1.9.3 and measured its performance under two different setups described below.

6.1 Experimental Setup

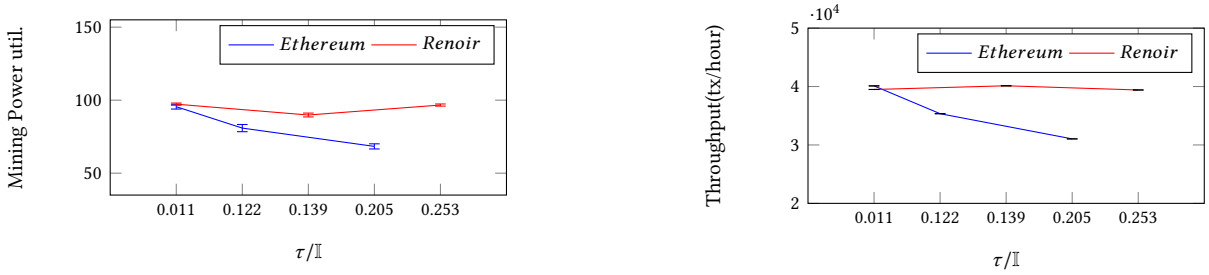
First, we deploy a Renoir equipped node to the Ethereum mainnet and investigate the extent of reduction in the block validation time due to Renoir. The node had one 2.19GHz dual-core CPU, 8 GB RAM, and 6.4TB NVMe SSD.

Second, we create a private blockchain network on 50 Oracle Virtual Machines to observe the effect of varying *block creation time* on both Ethereum and Renoir. In this setup each VM is equipped with one 2.19GHz dual-core CPU, 8 GB RAM and 128GB HDD. All VMs were running ubuntu 16.04 with download and upload bandwidth of 1 GBps and 100 MBps, respectively. Each VM runs one blockchain network node. Throughout the experiment, we have controlled the block mining difficulty so as to take 15 seconds to solve the Proof-of-Work puzzle to mine the block. This implies that the average block inter-arrival in our Ethereum experiment is 15 seconds + block propagation delay + block creation time + block validation time and the last quantity is replaced with the Renoir validation time for Renoir. With this experimental setup, we compare Renoir and Ethereum on metrics we define below.

Nodes, network delays and topology. We assign mining power to each node in our 50 node setup in accordance with the distribution of mining power of the top 50 miners of the Ethereum network [2]. The top 50 miners (by mining power) contribute to around 99.98% of total mining power of the real Ethereum network, with the most powerful miner controlling ~33% of the total mining power. Also, we use the geographical location of these top 50 Ethereum miners from [2] to mimic the location of our 50 nodes. We ensure that the inter-node latency (using Linux `tc` command) between any pair of nodes is in line with the ping delay corresponding to the geographic locations of the nodes [1] in effect mimicking the delays between real nodes of the Ethereum mainnet.

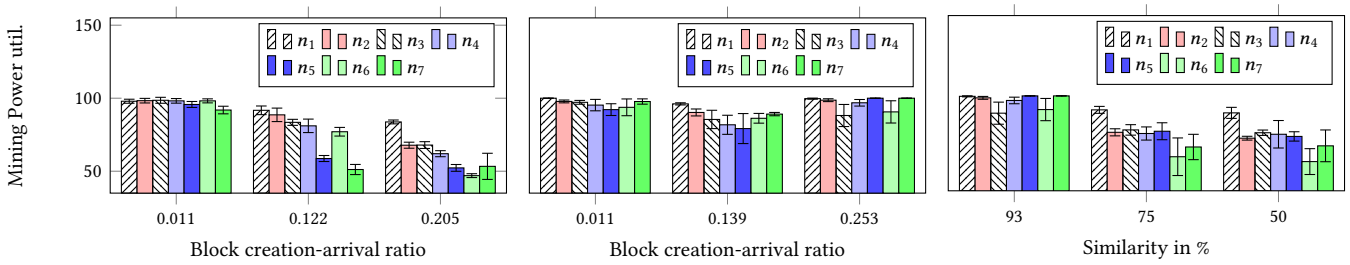
In line with the topology of the Bitcoin network where the degree of a node follows the power-law distribution [10] we design the topology of our experimental setup as follows: Each node connects to a random set of other nodes such that the degree of the node follows the power-law distribution.

Applications tested. We evaluate both Renoir and the Ethereum by deploying three types of smart contracts, each implementing quicksort, 2D matrix multiplication, and loop iteration with basic arithmetic operations. Throughout the experiment, we maintain an



(a) MPU (b) Throughput

Figure 9: Mining power utilization and throughput of Ethereum and Renoir with a confidence interval of 95% measured in our private network with varying block creation-arrival ratio



(a) Ethereum (b) Renoir (c) Renoir, Block creation-arrival ratio=0.253

Figure 10: Mining power utilization of the first 7 nodes of Ethereum and Renoir with a confidence interval of 95% measured in our private network with varying block creation-arrival ratio and varying similarity

average of ~165 transactions per block which is the average number of transactions in a Ethereum block. Thus, whenever required, we vary the block creation time by varying the time it takes for a node to execute each of these transactions.

Parameters and Metrics. The *block creation-arrival ratio* is the ratio of the block creation time to the average block inter-arrival time (block creation time+block validation time+avg. block mining time) and is an important parameter in the performance evaluation of a blockchain system. In particular, a high block-creation interval ratio indicates that the system has high throughput if it(high block-creation interval) is the result of including more number of transactions in the block. Thus, we investigate the effect of increasing block creation-arrival ratio on *mining power utilization*, block validation time and throughput. Throughput is the number of main chain transactions processed per unit time. Mining power utilization of a node is the ratio of the number of blocks mined by the node that eventually makes it to the main chain to that of the total number of blocks mined by the node. This indicates the extent to which mining was successful - the blocks that do not make it to the main chain represent wasted effort.

6.2 Experiments and Results

Reduction in block validation time. Figure 11 illustrates the reduction in block validation time as a result of using Renoir in Ethereum public network. Specifically, observe that without Renoir a Ethereum host node takes ~200 milliseconds to validate a received block, whereas a node equipped with Renoir only takes ~100 milliseconds, hence, a 50% reduction in block validation time. The reduction in block validation time is lower than our estimated similarity of more than 80% (§3.3), because the node spends additional time to decide whether to skip a transaction or not. Similarly, Figure 12 illustrates the reduction in block validation time we observe on our second setup (private 50 node network) with varying block creation-interval ratio with ~93% similarity. In particular, we find that for high block creation-interval ratio, nodes equipped with Renoir only spend a tiny fraction of a second to validate received block. On the other hand, nodes equipped with Ethereum takes over a few seconds to validate the block.

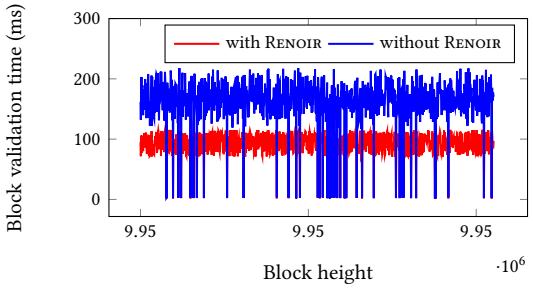


Figure 11: Validation time of 1000 real Ethereum blocks at two nodes, one of which is equipped with Renoir and the other is not.

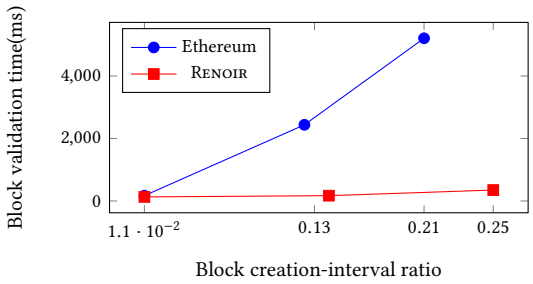


Figure 12: Time taken by the node to validate a received block with and without Renoir for varying block creation-interval ratio.

Mining power utilization. A low mining power utilization indicates a high degree of wasted computation. In Figure 9(a), we observe that the overall mining power utilization significantly drops with increase in block creation-arrival ratio, i.e., with high block creation/validation time. In Figure 10(a), we observe that with the increase in block creation-arrival ratio, the mining power utilization of the first seven nodes in Ethereum drops significantly. Furthermore, this decrease is more for nodes with lower mining power. The reason behind this decrease is that, with high block creation and validation time, blocks take longer to propagate as nodes only forward those blocks for which it has validated all its ancestor blocks. This results in a high fork rate in the network and hence low mining power utilization.

In Figure 9(a) we observe that mining power utilization of the system in Renoir remains unaffected even for high block creation-interval ratio of 0.253, unlike Ethereum. Figure 10(b) illustrates the mining power utilization of first seven Renoir nodes (ordered by mining power) in our experiment with 93% similarity. Unlike Ethereum, in Renoir the mining power utilization of nodes remains unaffected. This is due the fact that despite high block creation time, as illustrated in Figure 12 the block validation time in Renoir is very small. We also evaluate Renoir by varying the similarity and measure its effect on mining power utilization for block creation-interval ratio of 0.253. Figure 10(c) illustrates our findings. Observe that even with 50% similarity, mining power utilization of nodes are better than mining power utilization of Ethereum at higher block creation-arrival ratio of 0.205. This illustrates that Renoir achieves better, mining power utilization and is robust against variations in similarity.

Throughput Figure 9(b) illustrates that throughput of the system in Ethereum declines with the increase in block creation-arrival ratio. On the other hand, we observe that higher block creation-interval ratio barely affects the throughput of Renoir. This is because of the higher fork rate, as observed in figure 9(a), which then delays the extension of the main chain. Note that the increase in block creation-interval ratio in our experiment is a result of the inclusion of the transactions that require more amount of computation, in the blocks and not the increase in the number of transactions. Later will give the scope to increase the throughput further but, at the cost of an increase in block size.

7 RELATED WORK

We present related work in three parts. The first is about other measurement studies on the Ethereum network. We then look at efforts that have attempted to reduce the block validation time. Finally, we briefly discuss the recent proposals on smart contract scalability and how Renoir can complement these proposals for better performance.

Blockchain measurements. Gencer et al. [20] measure the degree of decentralization in the Ethereum network. They observed that Ethereum nodes are more widely distributed than Bitcoin nodes but have less spare bandwidth. Kim et al. [23] explored the RLPx and DEVp2p layer of Ethereum network. Ethereum uses RLPx for node discovery, and DEVp2p to explore the information propagation in the network. Wang et al. [33] investigate the behavior of participants in mining pools and their effects on the Bitcoin's transaction fees and propagation delay of transactions. TxProbe [16] explores the Bitcoin network and propose a technique to reconstruct Bitcoin topology with precision and recall surpassing 90%. To our knowledge, there has not been any study that attempted to measure or estimate the set of transactions contained in the transaction pool at nodes and their dependencies with each other, which is critical in reducing validation time as we have seen.

Concurrent smart-contracts. In an alternative approach, a new line of work has tried to reduce the block validation time by concurrently executing transactions [7, 17, 37]. Dickerson et al. [17] enable the miner to concurrently execute the transaction using a pessimistic abstract lock and inverse-log represented as a directed acyclic graph (happen-before graph). This inverse-log is later used in the validation phase, to replay the block creator's parallelization schedule. Anjana et al. [7] replaced the pessimistic lock with OCC favoring low-conflict workloads but at the cost of high abort rate for transactions with higher conflicts. Zhang et al. [37] improves concurrency of the validation phase by recording the write set of each transaction in the block, at the cost of additional storage and communication overhead. Due to dependency between transactions, they lead to wastage of computation resources and also have limited scalability. Furthermore, these approaches rely on a large amount of additional resources for parallel execution of transactions

Smart contract scalability. Recently, numerous works have proposed mechanisms to enable computationally intensive transactions in blockchains [13, 15, 18, 22, 32]. Typically, these approaches delegate the task of intensive computation to a set of offline, untrusted volunteer nodes and later run a result aggregation protocol to identify the correct execution result. These approaches do not address the problem of reducing the block validation time. Nevertheless, due to the modularity of our design, Renoir can be used as a complementary system to avoid re-execution of transactions during the result aggregation protocol.

8 SUMMARY & EXTENSIONS

In this paper, we presented Renoir, whose design enables nodes to skip the (re)execution of transactions in the block validation phase to speed the validation. Via a comprehensive empirical study of block and transaction propagation in Ethereum mainnet, we have determined that the scope of reduction in block validation time is

significant. Renoir uses information about the read and write sets of transactions to decide about skipping executing the transaction. We experimentally evaluated the Renoir on Ethereum mainnet and our 50 node network setup and determined that Renoir outperforms Ethereum especially for high block creation-arrival ratios. Lastly, we have presented a complete theoretical proof of correctness of Renoir.

Although we present an implementation of Renoir on permissionless PoW based blockchains, the core ideas of Renoir will work just as well with other consensus protocols such as committee based BFT [6, 9, 24, 30, 35], Paxos [25], RAFT [29]. Furthermore, as permissioned blockchains usually have fewer nodes and the connectivity among these nodes are more regulated, we expect to observe a higher value of similarity, than with permissionless blockchain networks. However, in a typical committee based protocol, only the leader creates the block which other nodes validate. Thus to extend Renoir to committee based consensus, each node will execute the transactions available in their transaction pool and cache the result of the execution. On receiving the block from the leader, the node will use the cached result by following the Renoir mechanism.

ACKNOWLEDGMENTS

We thank Shashi Bhushan Singh for his generous help in the data collection part. We also thank Oracle Corp. for supporting us with a grant in terms of free cloud credit, which we used to run all our experiments demonstrated in this work.

REFERENCES

[1] [n.d.]. Global Ping Latency. https://wondernetwork.com/pings [Online; accessed 15-March-2020].
[2] [n.d.]. Mining Power Distribution of Ethereum. https://investoon.com/charts/mining/eth [Online; accessed 16-May-2019].
[3] 2020. Ethereum clients. https://github.com/ethereum/wiki/wiki/Clients,-tools,-dapp-browsers,-wallets-and-other-projects#ethereum-clients
[4] 2020. Go Ethereum, Official Go implementation of the Ethereum protocol. https://geth.ethereum.org/
[5] Maher Alharby, Roben Castagna Lunardi, Amjad Aldweesh, and Aad van Moorsel. 2020. Data-Driven Model-Based Analysis of the Ethereum Verifier's Dilemma. arXiv:2004.12768 [cs.CR]
[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.
[7] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 83–92.
[8] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2018. Deconstructing the blockchain to approach physical limits. *arXiv preprint arXiv:1810.08092* (2018).
[9] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. State machine replication in the Libra Blockchain. *The Libra Assn., Tech. Rep* (2019).
[10] Annika Baumann, Benjamin Fabian, and Matthias Lischke. 2014. Exploring the Bitcoin Network. *WEBIST 2014 - Proceedings of the 10th International Conference on Web Information Systems and Technologies* 1. https://doi.org/10.5220/0004937303690374
[11] Vitalik Buterin et al. 2013. Ethereum white paper. *GitHub repository* (2013), 22–23.
[12] JPMorgan Chase. 2020. A permissioned implementation of Ethereum supporting data privacy.
[13] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 185–200.
[14] Sourav Das, Nitin Awathare, Ling Ren, Vinay Joseph Ribeiro, and Umesh Bellur. 2020. Better Late than Never; Scaling Computations in Blockchain by Delaying Transactions. *arXiv preprint arXiv:2005.11791* (2020).
[15] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. 2019. YODA: Enabling computationally intensive contracts on blockchains with Byzantine and Selfish nodes. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*.
[16] Sergi Delgado-Segura, Surya Bakshi, Cristina Pérez-Solà, James Litton, Andrew Pachulski, Andrew Miller, and Bobby Bhattacharjee. 2019. TxProbe: Discovering Bitcoin's Network Topology Using Orphan Transactions. In *Financial Cryptography and Data Security*, Ian Goldberg and Tyler Moore (Eds.). Springer International Publishing, Cham, 550–566.
[17] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 303–312.
[18] Jacob Eberhardt and Stefan Tai. 2018. ZoKrates-Scalable Privacy-Preserving Off-Chain Computations. In *IEEE International Conference on Blockchain. IEEE*.
[19] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 281–310.
[20] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. 2018. Decentralization in Bitcoin and Ethereum Networks. *CoRR* abs/1801.03998 (2018). arXiv:1801.03998 http://arxiv.org/abs/1801.03998
[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 51–68.
[22] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1353–1370.
[23] Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Measuring Ethereum Network Peers. In *Proceedings of the Internet Measurement Conference 2018* (Boston, MA, USA) *(IMC '18)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/3278532.3278542
[24] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {usenix} security symposium ({usenix} security 16)*. 279–296.
[25] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.
[26] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870* (2018).
[27] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. 2015. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 706–719.
[28] Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[29] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro
[30] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. 2017. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 426–438.
[31] Parity Technologies. 2020. Parity Ethereum Client - OpenEthereum. https://www.parity.io/ethereum/
[32] Jason Teutsch and Christian Reitwießner. 2017. A scalable verification solution for blockchains. (2017).
[33] Canhui Wang, Xiaowen Chu, and Qin Yang. 2019. Measurement and Analysis of the Bitcoin Networks: A View from Mining Pools. *CoRR* abs/1902.07549 (2019). arXiv:1902.07549 http://arxiv.org/abs/1902.07549
[34] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.
[35] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.
[36] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. [n.d.]. OHIE: Blockchain Scaling Made Simple. In *2020 IEEE Symposium on Security and Privacy (SP)*. 112–127.
[37] An Zhang and Kunlong Zhang. 2018. Enabling concurrency on smart contracts using multiversion ordering. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data*. Springer, 425–439.