Vegeta: Enabling Parallel Smart Contract Execution in Leaderless Blockchains

Tianjing Xu and Yongqi Zhong, *Shanghai Jiao Tong University;* Yiming Zhang, *Shanghai Jiao Tong University and Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3;* Ruofan Xiong, *Xiamen University;* Jingjing Zhang, *Fudan University;* Guangtao Xue and Shengyun Liu, *Shanghai Jiao Tong University and Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3*

https://www.usenix.org/conference/nsdi25/presentation/xu-tianjing

This paper is included in the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Vegeta: Enabling Parallel Smart Contract Execution in Leaderless Blockchains

Tianjing Xu[1*], Yongqi Zhong[1*], Yiming Zhang[1,2*], Ruofan Xiong[3], Jingjing Zhang[4], Guangtao Xue[1,2], Shengyun Liu[1,2#]

[1]*Shanghai Jiao Tong University* [2]*Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3* [3]*Xiamen University* [4]*Fudan University*

Abstract

Consensus and smart contract execution play complementary roles in blockchain systems. Leaderless consensus, as a promising direction in the blockchain context, can better utilize the resources of each node and/or avoid incurring the extra burden of timing assumptions. As modern Byzantine-Fault Tolerant (BFT) consensus protocols can order several hundred thousand transactions per second, contract execution is becoming the performance bottleneck. Adding concurrency to contract execution is a natural way to boost its performance, but none of the existing frameworks is a perfect fit for leaderless consensus.

We propose *speculate-order-replay*, a generic framework tailored to leaderless consensus protocols. Our framework allows each proposer to (pre-)process transactions prior to consensus, better utilizing its computing resources. We instantiate the framework with a concrete concurrency control protocol Vegeta. Vegeta speculatively executes a series of transactions and analyzes their dependencies before consensus, and later deterministically replays the schedule. We ran experiments under the real-world Ethereum workload on 16-vCPU virtual machines. Our evaluation results show that Vegeta achieved up to 7.8× speedup compared to serial execution. When deployed on top of a leaderless consensus protocol with 10 nodes, Vegeta still achieved 6.9× speedup.

1 Introduction

Blockchain or state machine replication (SMR) is a promising solution for providing trustworthy services among a group of untrusted entities or *nodes*. To mask failures, blockchain systems rely on a consensus protocol [32, 50, 51, 58] to reach an agreement on the sequence of blocks (and transactions). Each node is abstracted as a state machine [65], which sequentially executes the ordered transactions based on the same initial *world state*. Towards wide-scale adoption, boosting performance of blockchain systems is becoming a hot topic in both academia and industry [36, 59, 68, 77].

As Nakamoto-style consensus (such as Bitcoin [58] and Ethereum 1.0) has limited performance, modern permissionless and permissioned blockchains [17, 24, 37] are adopting Byzantine-Fault Tolerant (BFT) consensus protocols [22, 25, 29, 49, 57, 78] as their consensus layer. Most classical BFT protocols are leader-based [22, 25, 39, 78], meaning that a single leader is in charge of disseminating proposals and coordinating consensus. To select a single and stable leader, however, they rely on additional timing assumptions which may not be realistic in large-scale Wide Area Networks (WANs). Besides, such a single node may easily become the performance bottleneck [47, 68, 76].

Recently, many leaderless protocols are proposed targeting deployment in blockchain systems. Leaderless variants either have no timing assumption to elect any special role [17, 29, 41, 55, 57, 60], or assign the leader role to every node [28, 68]. Each node is required to act as a proposer and make a block proposal, and the final decision is a (sub)set of all proposals. Leaderless protocols are becoming an appealing solution to blockchain systems. For instance, Sui [7, 38], HashGraph [17] and Conflux [52] use a DAG-based protocol as their consensus layer, while AntChain leverages an asynchronous consensus for adaptive advancement [55].

BFT consensus protocols nowadays can process hundreds of thousands of transactions per second (TPS) [29], or even beyond one million [19] when deployed within a LAN. In contrast, the serial transaction execution engine of Ethereum platform only achieves around 100 TPS throughput [44], throttling the whole system. A natural way to improve efficiency of smart contract execution is to leverage modern multi-core architecture and add concurrency to transaction processing [26, 30, 36, 53].

When designing a concurrency control protocol in blockchain contexts, we argue that the consensus layer should also be taken into consideration as they play complementary roles. Currently, there are two trends in designing such pro-

[*]Equally contributed authors.
[#]Corresponding author: shengyun.liu@sjtu.edu.cn

tocols [66]: the *order-execute* framework [15, 36, 48], where transactions are (concurrently) executed after consensus in a deterministic way and produce the same results; and the *execute-order-validate* framework [14, 42, 46], where transactions are first speculatively executed by proposer(s) before consensus, and (re-)executed and/or validated by other nodes after consensus. In the order-execute framework, transaction execution is totally decoupled from consensus and thus can be readily plugged into any consensus protocol (leader-based or leaderless). The execute-order-validate framework allows proposer(s) to orchestrate transaction execution before consensus, such that other nodes can follow the order produced by proposer(s) and execute them in an efficient manner.

Despite being a more promising direction towards blockchain deployments, leaderless consensus introduces new challenges for efficient transaction processing, and neither of the aforementioned frameworks is a perfect fit. On the one hand, the order-execute framework cannot fully inherit the scalability feature from leaderless consensus, as each node must execute every block after consensus. Besides, they must schedule transactions in a deterministic way in order to produce the same execution results, limiting speedups when inconsistency resolution is frequently involved. On the other hand, the execute-order-validate framework either targets leader-based consensus [15, 42, 44, 46], or aborts transactions if new conflicts are introduced after consensus [14]. In the second case, aborted transactions must go through the whole procedure (i.e., execute, order and validate) again. Besides, neither takes leaderless consensus into special consideration.

We observe that the transaction execution layer of any blockchain system or SMR protocol can be divided into two parts: the pre-consensus part that allows every node to individually process distinct transactions, and the post-consensus part that requires every node to perform the same set of tasks (e.g., transaction execution and validation). To better adapt to leaderless consensus and improve scalability, in general we should place more work in the former if this can effectively reduce the burden of the latter.

Notably, the idea of allowing nodes to (pre-)process transactions prior to replication has been extensively explored in deterministic databases [40, 45, 69, 73, 79]. For instance, for transactions which cannot determine read/write sets a priori (also named dependent transactions), reconnaissance queries [72] is typically performed before replication. This pattern is a natural fit for transferring the scalability feature of leaderless consensus to the smart contract layer and thus deserves further study.

We accordingly propose a *speculate-order-replay* framework for efficient smart contract execution in leaderless blockchains. Before disseminating block proposals, each node first speculatively executes a series of transactions and/or performs any task that may help accelerate post-consensus tasks. Once blocks are totally ordered, each node enters the *replay* phase, during which it tries to (efficiently) execute transactions with the help of the information provided by speculation. Because in leaderless consensus there exist other concurrent proposals, such information may not be accurate with regard to the actual execution context. In case some transactions/operations introduce new conflicts in the replay phase, they must be processed in a deterministic way.

We instantiate this framework by a concrete concurrency control protocol (Vegeta). Vegeta realizes transaction-level parallelism, meaning that each transaction is treated as an indivisible unit. In the speculation phase, Vegeta executes transactions in a fully parallel manner, and obtains their read/write sets in order to construct a DAG that encapsulates the dependency information. In the replay phase, each node tries to "replay" the schedule provided by speculation. If a transaction *tx* produces the same read/write sets, its execution is finalized. Otherwise, if *tx* introduces new dependencies by accessing new keys, *tx* is re-executed after other transactions complete. The rationale of Vegeta is simple: although proposers are less likely to obtain correct execution results based on an inaccurate world state, the dependency information provided is most likely accurate.

We have implemented Vegeta in Golang and evaluated it with the Ethereum workload, meaning all the transactions and blocks are real and taken from the most representative smart contract platform. We ran experiments on Amazon EC2 platform, both on a single virtual machine and among a group of at most 10 machines in order to evaluate Vegeta with a leaderless consensus protocol. The evaluation results demonstrate that Vegeta on a single machine achieved an average speedup of $7.8\times$ compared to serial execution. We also evaluated the performance of AriaFB [56], a deterministic optimistic Concurrency control protocol. Vegeta outperformed AriaFB by up to $2.1\times$. We postpone the proof of Vegeta to Appendix A due to space limit.

2 Background and Related Work

2.1 System Model

We focus on the blockchain or state machine replication (SMR) problem [65], a solution to which is a highly reliable and available distributed system. Each node is abstracted as a state machine and sequentially executes a series of requests or *transactions* issued by clients. We target authenticated Byzantine failure model [51], meaning that the faulty nodes may exhibit any malicious behavior except for breaking cryptographic schemes. We assume at most f nodes can be faulty and there are a total of $n = 3f + 1$ nodes, the lower bound for BFT SMR protocols under asynchrony assumption [32].

Transaction semantics. Each transaction contains several operations and may read and/or write a set of keys or objects, which are collectively maintained as a world state in blockchains [5]. The world state can be simply considered a key-value store. Blockchain systems provide serializabil-

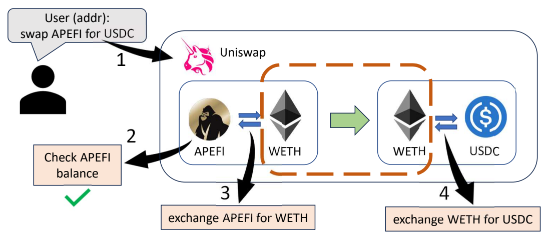


Figure 1: Example for Ethereum smart contract invocation.

ity [21] semantics, in the sense that any execution result is equivalent to the outcome of sequentially executing all transactions in some (pre-defined) order. To enable parallel processing, we further distinguish between conflicting and non-conflicting transactions. We say two transactions tx_1 and tx_2 are *conflicting* if they access the same key x and at least one of them writes x. Otherwise, tx_1 and tx_2 are *non-conflicting*.

2.2 Smart Contracts

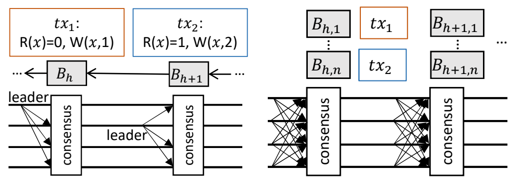
Smart contracts [71] implement application logic and can be simply considered a piece of code deployed on top of blockchains. Each transaction invokes the code of a contract, which may recursively invoke another contract.

In Figure 1 we give an example for Ethereum smart contract invocation. This example [2] is stripped off from the Ethereum mainnet. A user wanted to trade some Ape Finance tokens (APEFI) [1] for USDC [9], a stablecoin pegged to US dollar. Uniswap [8] is a Decentralized Exchange (DEX) contract for facilitating token swap. The user first invoked the Uniswap contract (step 1), which then checked the balance of the user by invoking the APEFI contract (step 2). At that particular moment, there was not a liquidity pool available for exchanging APEFI and USDC. Hence, the Uniswap contract leveraged Wrapped ETH (WETH) [10] as an intermediary token for achieving the desired trade (step 3). Finally, the Uniswap contract exchanged WETH for USDC (step 4). By this example we can observe, when a user initiates a transaction, the user is not aware of the (full) list of addresses (or keys) that the transaction will access. To accurately obtain such information, an Ethereum node must execute this transaction based on the most recent world state.

2.3 Leader-Based vs. Leaderless Consensus

Existing SMR protocols can be roughly classified into two categories according to the ways they order proposals: leader-based and leaderless. Leader-based protocols rely on a special role (named leader) to coordinate consensus, i.e., disseminating proposals to and/or collecting votes from other nodes.

We take PBFT [25] and HotStuff [78], two representative leader-based BFT protocols for example. PBFT assigns the leader role to a single node until this node is faulty or partitioned, upon which a view change sub-protocol is triggered



(a) Leader-based (b) Leaderless

Figure 2: Comparison between leader-based and leaderless consensus protocols.

to elect a new leader. HotStuff proactively rotates the leader role, ensuring that every node has the opportunity to propose. Upon making a new block proposal B_{h+1} (see Figure 2a), the leader is aware of the most recent proposal B_h that precedes B_{h+1}. Transitively, the leader knows all the previous proposals. Thus, the leader can pre-execute tx_2 based on the execution result of tx_1, which is included in B_h. In summary, leader-based protocols have the following feature:

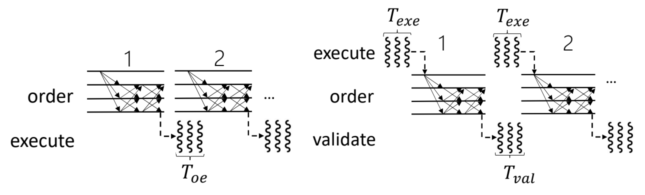
Definition 1. *(Perfect Context) we say a consensus protocol has perfect contexts, if when any node makes a proposal $\mathcal{B}$, the node is aware of an ordered list of all proposals that are (or will be) committed before $\mathcal{B}$.*

Both PBFT and HotStuff and several permissionless blockchains such as Bitcoin [58] and Ethereum [5] have perfect contexts. With this feature, the proposer (e.g., the single leader) can (pre-)execute all transactions before consensus, give a total or partial order among them and insert (the digest of) execution results as well as transaction dependencies into the proposal. Other nodes, upon receiving the proposal, can (re-)execute the transactions in the same pre-defined order and obtain the same results.

Leaderless protocols, in contrast, have no fixed leader role but allow every node to act as a proposer. One type of leaderless protocols [16,28,68] pre-partitions sequence space and assigns them to every node, while another type [29,31,38,41,57] is designed under asynchrony assumption. The final output of leaderless protocols is a combination of all committed proposals. By allowing each node to act as a proposer, leaderless protocols can distribute workloads and better utilize resources (e.g., CPU and network capacity). Since leaderless protocols allow concurrent proposals made by multiple proposers, they have no perfect context. For instance, in Figure 2b, concurrent proposals $B_{h,1},\ldots,B_{h,n}$ are proposed by distinct nodes, and hence are totally ordered only after consensus. Their proposers have no perfect knowledge of other concurrent proposals and thus cannot get correct results.

2.4 Parallel Execution

To integrate parallel execution into blockchains or SMR protocols, existing approaches can also be categorized into two types: the order-execute and execute-order-validate frameworks, as shown in Figure 3.



(a) order-execute (b) execute-order-validate

Figure 3: Existing frameworks for parallel execution.

Order-execute. The order-execute framework [15, 36, 48] (Figure 3a) orchestrates a (parallel) execution phase that emerges after consensus. Each node individually tries to parallelize transaction execution in a deterministic way. Typically, the results of parallel execution must be equivalent to the ones obtained by sequentially executing those transactions in the order given by consensus. The order-execute framework "blindly" imposes constraints on the order of transactions and may incur additional costs for, e.g., sequentially executing non-conflicting transactions and/or re-executing conflicting ones. The execution component in order-execute is decoupled from consensus, and thus can be integrated into any consensus protocol (leader-based or leaderless). Saber [54] improves transaction execution by allowing different groups to execute different transactions. The idea of Saber is akin to inter-node parallelism and largely orthogonal to our contribution.

Execute-order-validate. The execute-order-validate framework [14, 18, 42, 46, 67] (Figure 3b), in contrast, enables proposer(s) to (pre-)execute transactions before consensus. There are two trends in instantiating this framework: they either target a leader-based consensus protocol [18, 42, 67] or resolve conflicts in a trivial way [14, 46]. In the former case, the single leader orchestrates transaction execution and re-orders them if necessary, while other nodes follow the schedule and accordingly execute transactions in the same pre-defined order. In the latter case, existing protocols tend to *trust* the execution results obtained prior to consensus, but resort to a heavy fallback mechanism to resolve newly-introduced conflicts. For example, Eve [46] assumes the read/write sets of each transaction can be obtained a priori, otherwise it must rely on a leader-based protocol (i.e., PBFT) to resolve conflicts. Hyperledger Fabric [14] requires every peer to speculatively execute transactions and later reach consensus also on execution results, so as to deal with non-deterministic operations. If new conflicts are introduced after consensus, conflicting transactions must go through the whole procedure again.

Deterministic database systems. Concurrency control has long been a hot topic in database systems [13, 21, 64]. There are numerous works targeting deterministic scheduling and execution [12, 33, 34, 43, 72–74]. To enable parallel execution while still producing identical results, deterministic databases usually require the reach/write sets of each transaction can be obtained a priori. Otherwise, reconnaissance queries [33, 61, 62, 72, 73] or speculative execution should be performed before transaction scheduling. Such a technique [72] decomposes a *dependent transaction* with unknown read/write sets into a combination of some read-only transactions and a conditional-write transaction. The read-only transactions discover the read/write sets (say, before replication),while the conditional-write transaction commits (after replication) if the read values match. Calvin [73] adopts a lock-level concurrency control protocol and orchestrates lock acquisition based on a pre-defined total order. Calvin does not readily support dependent transactions, but can leverage the mechanism discussed above (also named Optimistic Lock Location Prediction, OLLP) to obtain read/write sets. Caracal [62] and Epic [61] utilize multi-version concurrency control (MVCC) to reduce conflicts and can also run a similar identification phase to determine read/write sets. Reconnaissance query is analogous to the speculation in Vegeta, but Vegeta and our framework specifically target leaderless blockchains (under Byzantine faults) and are more focused on co-design between pre- and post-consensus tasks. Besides, Vegeta does not consider the speculation output as potential results, but only relies on it for dependency analysis.

Aria [56] does not require read/write sets to be known before execution but first executes transactions batch by batch in a fully parallel manner. If any transaction *tx* introduces conflicts with an earlier transaction in the same batch, *tx* is aborted and re-tried with the next batch. When a workload results in a very high abort rate, Aria further resorts to a fallback mechanism (e.g., Calvin [73]) to deterministically resolve the conflicts, instead of repeatedly trying aborted transactions. In a sense, Vegeta uses a similar batching mechanism in the replay phase, but further leverages the speculation phase to wisely group transactions into batches. Morty [23] is a recently-proposed concurrency control method that leverages transaction re-execution to improve throughput. Our method instead introduces re-execution to ensure consistency among different nodes. Basil [70] is a newly-proposed BFT transactional storage system, which demonstrates that the leaderless feature can effectively improve performance.

3 The Speculate-Order-Replay Framework

We observe that the successful story of leaderless consensus protocols stems from the equality of all nodes in disseminating proposals and processing transactions [75], thus effectively balancing workloads. We intend to further achieve such a feature in the contract execution layer. To this end, we study a generic framework called speculate-order-replay (SOR, as shown in Figure 4). Similar to utilizing resources of all nodes in a leaderless consensus protocol, the speculate-order-replay framework allows each node to process and speculatively execute distinct transactions before consensus. We call this step *speculation*. Although proposers do not have a perfect execution context, they may still obtain crucial information about transaction conflicts, read/write sets and program traces,

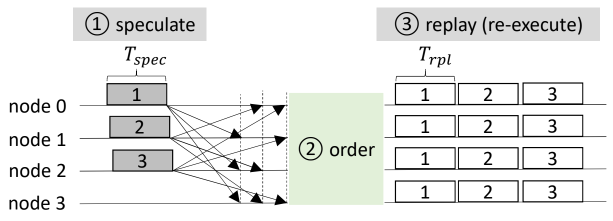


Figure 4: The speculate-order-replay framework.

which can greatly help maximize parallelism.

Once the speculation phase completes, each proposer takes the transactions and other information as input and enters the *ordering* phase. The ordering phase is run solely by a consensus protocol, and thus we omit its detail in this paper. Once the sequence of committed proposals is agreed upon by consensus, each node enters the *replay* phase. In general, conflicting operations must be executed in the same (serial) order, while non-conflicting operations can be processed in parallel. If the information provided by speculation is almost accurate, nodes in the replay phase may execute transactions with a maximum degree of parallelism, while at the same time paying no/few extra costs for handling (unnecessary) aborts. However, if some transactions impose new conflicts, they must be carefully treated. We postpone the detail to §4 as it is related to specific parallelism schemes.

Analysis of speedups. We give a simple analysis for demonstrating why the SOR framework may scale better with regard to the speedup achieved. The results are also shown in Table 1. We assume on average, sequentially executing a block of transactions takes T_{seq} seconds, which is the baseline for further comparison. Executing m blocks takes mt_{seq} seconds. Similarly, the order-execute framework (Figure 3a) takes mT_{oe} seconds to execute m blocks if each block on average costs T_{oe} seconds. Compared to serial execution, the speedup achieved by the order-execute framework is $\frac{T_{seq}}{T_{oe}}$.

As for the execute-order-validate framework (Figure 3b), each block first goes through an execution phase, which takes T_{exe} seconds. After consensus, each node validates the execution results by (re-)executing them, which takes T_{val} seconds. Thus, each block incurs $T_{exe}+T_{val}$ seconds for execution. Due to pipelining techniques, the execution phase of B_{h+1} can be overlapped with the validation phase of B_h. By doing so, the speedup approaches $\frac{T_{seq}}{T_{exe}}$ if m is large and $T_{exe} \geq T_{val}$. The latter assumption holds because the leader usually takes more time to orchestrate transaction execution [15].

Finally, we assume in SOR, the speculation phase takes T_{spec} seconds, while the replay phase takes T_{rpl} seconds (Figure 4). We further assume transactions are equally distributed to each node, which is an ideal assumption for SOR and in practice can be implemented by, e.g., applying load-balancing [68]. For each block, the speculation phase is only executed by one node, and different blocks can be speculatively executed in parallel. Thus, each block still takes $T_{spec}+T_{rpl}$ seconds, but the speedup, which takes m blocks into consideration, is $\frac{mT_{seq}}{mT_{spec}/n+mT_{rpl}} = \frac{T_{seq}}{T_{spec}/n+T_{rpl}}$.

Table 1: Analysis of speedups of different frameworks. $T_{oe} \approx \frac{T_{seq}}{\#cores} \approx T_{exe} \approx T_{val}$ if very few conflicts between transactions exist (which is not the case for Ethereum workloads). In contrast, $T_{rpl} \approx \frac{T_{seq}}{\#cores}$ if the information provided by the speculation phase is almost accurate and the degree of parallelism is approximately equal to the number of cores (though with many conflicts).

	latency	speedup
sequential	T_{seq}	-
order-execute	T_{oe} (Figure 3a)	$\frac{T_{seq}}{T_{oe}}$
execute-order-validate	T_{exe}+T_{val} (Figure 3b)	$\frac{T_{seq}}{T_{exe}}$
speculate-order-replay	T_{spec}+T_{rpl} (Figure 4)	$\frac{T_{seq}}{T_{spec}/n+T_{rpl}}$

We can observe that the bottleneck of speculate-order-replay mainly locates at the replay phase, which must be performed by every node. Intuitively, we may (moderately) put additional efforts into speculation if T_{rpl} can decrease. In contrast to order-execute, nodes in our framework can freely choose in what order transactions are executed, thus maximizing parallelism and alleviating re-execution burden. Compared to execute-order-validate, our solution is tailored to leaderless protocols and may better scale as nodes can speculate in parallel.

The SOR framework can be instantiated on different layers, e.g., transaction-level parallelism (Vegeta), lock-level parallelism or instruction-level parallelism. One may even integrate other techniques (e.g., zero-knowledge rollups [11] or I/O prefetching [27]) into SOR, so long as they help alleviate the burden of replay and the extra costs of speculation are acceptable.

4 Vegeta

We then elaborate on Vegeta, which follows the speculate-order-replay framework and implements transaction-level parallelism. Vegeta does not need to maintain a multi-version key-value store, nor does it need to roll back any transaction.

4.1 Speculation

The main purpose for the speculation phase is to give a specific (partial) order among transactions within a block, such that by following this order each node (in the replay phase) can efficiently execute them. A primary advantage of processing transactions prior to consensus lies in the absence of (unnecessary) ordering constraints between transactions. In

Algorithm 1: Speculation phase

```
   Variables: BLOCK: a block containing a list of transactions
              DAG: a two-dimensional array storing dependencies
              $access$: a map from key to $tx\_chain$
1  procedure Speculate():
2      $txs \leftarrow$ BLOCK.$transactions$
3      ParallelExecute($txs$)
4      $tx\_chains \leftarrow$ SortDependencyChains($txs$)
5      $txs \leftarrow \emptyset$
6      for $chain \in tx\_chains$ do
7          for $tx \in chain$ do
               /* Adjust the order of $txs$ */
8              if $tx \notin txs$ then
9                  append($txs$, $tx$)
10     BLOCK.$transactions \leftarrow txs$
11     DAG $\leftarrow$ BuildDAG($txs$)
12     call Consensus($\langle$BLOCK, DAG$\rangle$)
13 function ParallelExecute($txs$):
14     for $tx \in txs$ do
           /* Run in parallel mode */
15         $(tx.RS, tx.WS) \leftarrow$ execute($tx$)
16 function SortDependencyChains($txs$):
17     for $tx \in txs$ do
18         for $\forall\, key \in tx.RS \cup tx.WS$ do
19             append($access[key]$, $tx$)
20     for $chain \in access$ do
21         append($tx\_chains$, $chain$)
22     Sort $tx\_chains$ from the longest to the shortest
23     return $tx\_chains$
24 function BuildDAG($txs$):
25     for $tx \in txs$ do
26         for $\forall\, tx' : tx'.idx < tx.idx$ do
27             if $tx'.WS \cap tx.WS \neq \emptyset$ then
28                 DAG $[tx][tx'] \leftarrow WAW$
29                 continue
30             if $tx'.RS \cap tx.WS \neq \emptyset$ then
31                 DAG $[tx][tx'] \leftarrow WAR$
32             if $tx'.WS \cap tx.RS \neq \emptyset$ then
33                 if DAG $[tx][tx'] == WAR$ then
34                     DAG $[tx][tx'] \leftarrow WAW$
35                 else
36                     DAG $[tx][tx'] \leftarrow RAW$
37     return DAG
```

other words, each proposer can orchestrate an order at will, maximizing parallelism as much as possible.

On the contrary, any pair of conflicting transactions need to be processed sequentially (if they access the same key and one of them writes the key). Those conflicting transactions form a so-called dependency chain, which is the main enemy of high degree of parallel processing. The long dependency chain problem also appears in the Ethereum workload and was discussed in [35]. The heuristic here is we should give transactions within a longer dependency chain a higher priority, and thus in the replay phase the longer chain starts execution prior to other shorter chains. We thus have the following rule:

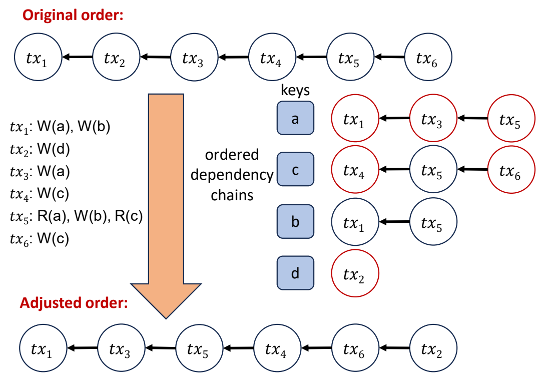


Figure 5: Example for the speculation phase. After obtaining the read/write sets, transactions are (re-)ordered according to the lengths of dependency chains. The corresponding DAG is depicted in Figure 6.

Rule 1. *Transactions should be sequenced in descending order based on the lengths of their dependency chains.*

The pseudocode is given in Algorithm 1. First, each node gets a series of transactions and executes them in parallel (Line 2-3). To obtain transaction dependencies, the proposer only needs the read/write sets of each transaction. Thus, at this stage transactions are executed in parallel and will not modify the world state. Based on the read/write sets, we are able to analyze transaction dependencies and create a schedule for the replay phase.

To this end, we first count the number of transactions that have accessed a given key and group them into a dependency chain (Line 17-21). A transaction may appear in several chains. Note that this simple method only obtains an approximation of accurate dependency chains, as each chain only takes one key for consideration. Nonetheless, this simple solution already leads to prominent performance (as shown in §6). Then, we sort different chains according to their lengths (Line 22). We finally arrange the transactions within a longer chain in front of those within a shorter chain and obtain an ordered list *txs* (Line 6-9).

Based on the order given in *txs*, we can analyze their dependencies and accordingly build a Directed Acyclic Graph (DAG) (Line 24-36). The dependency relationship forms a DAG (with no circle) because any transaction may only depend on ones with lower indices, but not vice versa. There are three types of dependencies: write after read (WAR), write after write (WAW) and read after write (RAW). If a transaction writes the same key that its previous transaction reads, the corresponding place of these two transactions in DAG is tagged with WAR. Other two types of dependencies are tagged similarly. Note that if two transactions have multiple dependencies, we prioritize WAW (Line 27-29) due to a specific optimization [56] that we will elaborate on §4.2. Also note that if two transactions have both WAR and RAW dependencies (for different keys), we treat them as having a WAW dependency (Line 34). Finally, the transactions, their read/write sets, as well as the DAG are inserted into a block proposal (Line 12). With such information, every node in the replay phase can process these transactions deterministically.

4.2 Replay

After consensus, blocks are totally ordered and each node enters the replay phase. The main purpose for replay is to deterministically and efficiently execute all transactions within each block. We only exploit intra-block parallelism, though enabling inter-block parallelism may further speed up execution, which we leave for future work. The replay phase is further divided into two stages: (1) the first stage that follows the DAG and execute transactions accordingly; and, (2) the second stage that re-executes transactions that may introduce new dependencies. For the first stage, our concurrency control method is inspired by Aria [56]. For the second stage, we first describe a scheme that simply re-executes all possibly affected transactions. In §4.3 we describe an optimization that further reduces re-execution rate.

The pseudocode is given in Algorithm 2. Each node gets a block, the read/write set of each transaction, as well as the DAG from the consensus component (Line 2). To enable parallel processing, transactions within a block are further divided into several *batches*, where in each batch all the transactions can be executed in parallel (if they do not access some new keys compared to the ones obtained in the speculation phase). Transactions within a block are executed in a batch-by-batch manner, and in each batch the execution results will not update the world state until all the transactions complete.

Each node repeatedly obtains a batch of transactions that have no *forward dependency* from the transaction list *txs* (Line 4), then executes them in parallel. We have the following specific rule for defining whether a transaction has no forward dependency:

Rule 2. *A transaction tx has no forward dependency if: (1) tx has no WAW dependency on any previous transaction that has not completed, and (2) tx does not have both WAR and RAW dependencies on any previous transactions that have not completed.*

The intuitions of Rule 2 are: (1) even if transaction *tx* WAR depends on tx', *tx* can start execution immediately because its update will not be visible to tx', as they are in the same batch; and (2) even if *tx* RAW depends on tx', *tx* can be re-ordered before tx' because the update of *tx*. The idea of Rule 2 is elaborated in [56].

Once *tx* finishes execution, we need to check its newly-generated read/write sets. If the read/write sets have changed

Algorithm 2: Replay phase

```
   Variables: $TxsRe$: a set of transactions to be re-executed
1  procedure Replay():
2      (BLOCK, DAG) ← Consensus()
3      $txs$ ← BLOCK.$transactions$
4      $ready$ ← PopTxsBatch($txs$)
5      while $ready \neq \emptyset$ do
6          for $tx \in ready$ do
               /* Run in parallel mode                    */
7              $(rs, ws)$ ← execute($tx$)
8              if $rs \neq tx.RS \lor ws \neq tx.WS$ then
9                  append($TxsRe$, $tx$)
10                 $ready \leftarrow ready \setminus \{tx\}$
11         for $tx \in ready$ do
12             commit($tx$)
13         $ready$ ← PopTxsBatch($txs$)
14 procedure ReExecute():
15     for $tx \in TxsRe$ do
16         execute($tx$)
17         commit($tx$)
18 function PopTxsBatch($txs$):
19     $ready\_txs \leftarrow \emptyset$
20     for $tx \in txs$ do
21         if $tx$ has no WAW-dependencies then
22             if no WAR $\lor$ no RAW then
23                 append($ready\_txs$, $tx$)
24                 $txs \leftarrow txs \setminus \{tx\}$
25     return $ready\_txs$
26 function commit($tx$):
27     Apply updates in $tx.WS$ to the world state
```

compared to the results obtained in the speculation phase, *tx* may introduce new dependencies with other transactions in the same block, and thus will be *re-executed* at the end of the replay phase (Line 8-10). Nonetheless, we can still commit other transactions in *ready* by updating the world state according to their write sets (Line 11).

Finally, we present a simple re-execution stage that happens at the end of the replay phase (Line 14-17). Transactions in $TxsRe$ are sequenced based on their indices and executed in serial order. In §A we prove that $TxsRe$ is the same among all correct nodes. In Figure 6 we depict an example for demonstrating the replay phase.

4.3 Reducing Re-Execution Rate

Because transaction re-execution may badly affect performance, we further improve our strategy for determining whether a transaction needs to be re-executed. The new strategy is described in Algorithm 3, which replaces Line 5-12 in Algorithm 2.

Remember that in Algorithm 2, a transaction should be re-

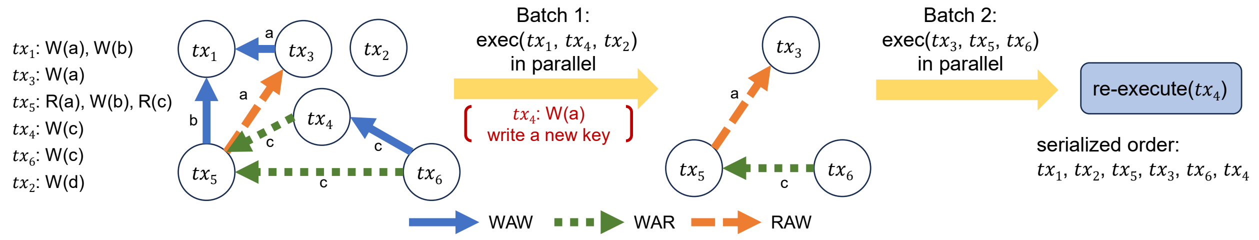

tx_1: W(a), W(b)
tx_3: W(a)
tx_5: R(a), W(b), R(c)
tx_4: W(c)
tx_6: W(c)
tx_2: W(d)
Batch 1: exec(tx_1, tx_4, tx_2) in parallel
tx_4: W(a) write a new key
Batch 2: exec(tx_3, tx_5, tx_6) in parallel
re-execute(tx_4)
serialized order: tx_1, tx_2, tx_5, tx_3, tx_6, tx_4
WAW
WAR
RAW

Figure 6: Example for the replay phase. On the left we show the read/write sets of each transaction obtained in the speculation phase. Batch 1 contains tx_1, tx_4 and tx_2 as they have no forward dependency (Rule 2). tx_4 newly writes key a, and thus will be re-executed at the end. Batch 2 contains tx_3, tx_5 and tx_6, even though tx_6 WAR depends on tx_5. The execution results are equivalent to executing tx_1, tx_2, tx_5, tx_3, tx_6 and tx_4 in serial order.

Algorithm 3: Improved replay strategy

```
1  $all\_keys \leftarrow$ all keys accessed by tx in $txs$ in speculation
2  while $ready \neq \emptyset$ do
3      for $tx \in ready$ do
           /* Run in parallel mode                    */
4          $(rs, ws) \leftarrow$ execute($tx$)
5          if $tx$ reads/writes a new key $key' \in all\_keys$ then
6              append($TxsRe$, $tx$)
7              $ready \leftarrow ready \setminus \{tx\}$
8              continue
9          if $tx$ reads a new key $key' \notin all\_keys$ then
10             append($read$, $tx$)
11         if $tx$ writes a new key $key' \notin all\_keys$ then
12             $new\_keys \leftarrow new\_keys \cup \{key'\}$
13     $read \leftarrow read \setminus \{tx : tx$ has not read any key $\in new\_keys\}$
14     commit($tx \in ready \setminus read$) based on DAG
15     for $tx \in read$ do
16         execute($tx$)
17         commit($tx$)
```

executed once its read/write sets have changed. In fact, some changes do not necessarily lead to re-execution, as they do not introduce new dependencies. For instance, if tx accesses fewer keys in the replay phase. We intend to further reduce re-execution rate, but at the same time keeping the method simple and straightforward to be implemented. To this end, we first group all the keys that have been accessed in the speculation phase (Line 1 in Algorithm 3). Then we further distinguish the following three cases:

1. If a transaction tx reads/writes a new key that other transactions also accessed during speculation (Line 5-8), it is highly possible that tx will introduce new dependencies. In this case, tx will be re-executed.

2. If tx reads a new key key' that other transactions did not access during speculation (Line 9-10, 15-16), key' may not necessarily introduce new dependencies, but tx (in $read$) has to wait until other transactions in the batch complete execution. If some transaction has newly written key' (in new_keys), tx will be executed again and get committed. Otherwise, tx is removed from $read$ and committed with other transactions by following the serial order extracted from DAG.

3. If tx writes a new key key' while other transactions did not access key' during speculation (Line 11-14), we record key' in new_keys. tx will also be committed with other transactions once all the transactions complete execution.

In other cases, tx will not be re-executed.

4.4 Further Discussion

This paper solely focuses on improving the performance of contract execution. In decentralized finance applications, however, transaction ordering is also crucial for participants' profits. Regarding the speculation phase in Vegeta, miners or proposers may be motivated to initiate frontrunning and sandwich attacks [3, 63] to increase their profits, rather than maximizing parallelism. To defend against adversarial players, existing literature either introduces additional restrictions on transactions ordering (e.g., by providing a fairness property [80]), or resorts to some economic mechanism (e.g., the proposer/block-builder separation [6]). For the former case, proposers cannot freely choose at what order transactions are replayed, thus limiting its performance gains. For the latter case, we may quantify the degree of parallelism and also leverage an incentive mechanism to motivate rational proposers.

5 Implementation

We have implemented Vegeta in Golang and adopted Go-Ethereum, a.k.a., Geth as our Ethereum execution environment. We integrated Ethereum Virtual Machine (EVM) [4] with Vegeta to enable transaction execution. We still used the data structure *StateDB* in EVM for caching and storing the world state in memory. *StateDB* can be simply considered an in-memory key-value store. As EVM only provides single-threaded execution, we modified *StateDB* in order to support shared access. To do so, we used a sync.Map structure

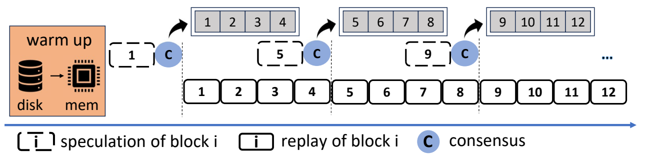


Figure 7: Example from node 1's perspective for consensus-integrated implementation ($n = 4$).

for thread-safe state management. We started a new EVM instance for each transaction, and all instances can concurrently access *StateDB*.

To focus on transaction execution performance, we skipped updating the Merkle Patricia Trie (MPT) in *StateDB* (after each execution). MPT is a combination of Patricia trie and Merkle tree. MPT is used to cryptographically authenticate and verify the content of the world state. One exception is we used MPT for correctness validation (§6.1), which updated MPT after the execution of each block. We also disabled data persistence to eliminate its impact on performance evaluation, allowing us to focus solely on transaction execution.

Consensus integration. In Vegeta the speculation phase of a given block B_h may overlap with the replay phase of some earlier block $B_{h'}$ ($h' < h$), so the two phases may compete for computing resources on each node. To study this impact, we also integrated a consensus component, which is a seminal leaderless consensus protocol proposed by Ben-Or, Kelmer, and Rabin (BKR) [20]. BKR is the core component of several recent works [31, 57, 77]. We chose BKR also because it allows us to easily distribute Ethereum blocks to each node, in such a way that the total order given by the consensus protocol is the same as given by Ethereum. In general, BKR requires each node to make a proposal in each consensus instance (i.e., for each sequence number). The final output of each instance is a subset of all proposals (ranging from $n - f$ to n proposals). To strictly follow the order of Ethereum workloads, we further restrict the number of proposals output by BKR to n, ensuring that no proposal is abandoned.

We assigned block B_h to a node i for speculation if $i = h\%n$, where n is the number of nodes[1]. We also configured a parameter K, where on each node i, if only K blocks are left for replay, node i takes the next block and starts its speculation phase. In principle, the smaller the K, the better the accuracy for speculation, as node i is with a more recent world state. However, to effectively pipeline the two phases of different blocks, K should not be too small. In our experiments, we set K to 2, as on average the time it takes to speculatively execute a block is less than twice the time it takes to replay a block (see §6.2). Before each test there is a warm-up phase, which loads the world state and all blocks after that state into memory. Figure 7 demonstrate how we integrate Vegeta into the consensus protocol. Our code is available at `https://github.com/Decentralized-Computing-Lab/Vegeta`.

[1]Note that in real-world cases, nodes continuously receive new transactions and batch them into blocks. No pre-defined order exists.

Table 2: Ethereum datasets used in our evaluation.

Tag	Date	Block range (height)	Tx count	Longest chain	Ratio
S_1	3/7/2023	16774645-16779644	739863	88136	8.39
S_2	3/7/2023	16774645-16777644	436115	52862	8.25
S_3	3/7/2023	16774645-16774745	15129	1779	8.50
S_4	11/17/2023-11/18/2023	18581726-18586725	747651	89961	8.31

6 Evaluation

We evaluated Vegeta under the real-world Ethereum workloads. We compared Vegeta with serial execution and a deterministic optimistic concurrency control protocol, AriaFB, which was derived from Aria [56]. AriaFB is a typical way of instantiating the order-execute framework. We adopted the same execution and commit phases used in Aria, and also integrated a fallback strategy (i.e., our replay phase) to get AriaFB, as vanilla Aria cannot handle high abort rates (see §6.2).

Testbed. We conducted experiments on Amazon EC2 platform. Our testbed consists of up to 10 m6i.4xlarge instances, with 3.5 GHz Intel Xeon Ice Lake 8375C processor of 16 vCPUs, 64 GB RAM, running Ubuntu 20.04 LTS. Every node has downloaded Ethereum workloads.

Datasets. Our evaluation was conducted mainly with the Ethereum datasets shown in Table 2. S_1 contains 5,000 blocks and is used to evaluate the performance in single-node mode and validate the correctness of Vegeta. S_2 contains 3,000 blocks and is used for multi-node test. S_3 contains 101 blocks and is used for time-consumption analysis of each phase (Figure 8 in §6.2). S_4 contains 5,000 more recent blocks and is used to evaluate the performance in both single-node and multi-node modes. In Table 2, tx count refers to the total number of transactions of all blocks. For each block, we first calculate the length of its longest dependency chain, i.e., the maximum number of transactions that have accessed a given key. Then, we sum the lengths of the longest chains of all blocks and get the longest chain in Table 2. Finally, the ratio equals to the tx count divided by the longest chain. The ratio approximately represents the maximum speedup Vegeta can achieve for a specific workload, if we assume every transaction has equal execution time. For simplicity, we did not change the block structure of Ethereum, but in practice we may only treat the workload as a transaction stream.

6.1 Correctness Validation

The correctness of Vegeta has been extensively validated as a by-product of our evaluation. This is because Ethereum

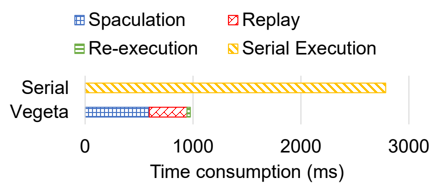


Figure 8: Time consumption of each phase.

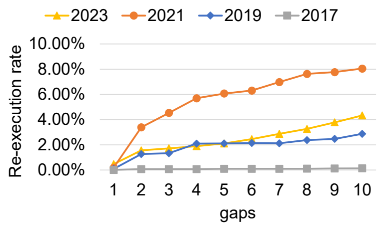


Figure 9: Re-execution rates.

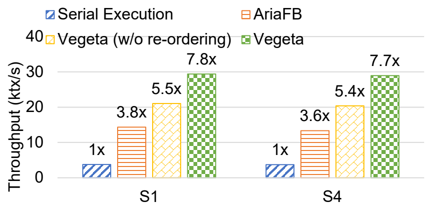


Figure 10: Throughput and speedups in single-node test.

maintains the world state as an MPT. Two states are identical if and only if the values of their MPT roots are equal. Every block carries a value of the state's MPT root after all the transactions in the block are executed. Although the execution result of Vegeta differs from the original state of Ethereum due to the re-ordering rule we adopted in the speculation phase, we can still perform the execution repeatedly (i.e., the speculation and replay phases) with the same series of blocks and validate whether their MPT root values are equal. We have managed to process a total of 5,000 blocks (S_1) carrying 739,863 transactions for 100 times, always resulting in a matching value of the MPT root in every block.

6.2 Single-Node Performance

We demonstrate the performance of Vegeta and other protocols when they were deployed on a single node.

Time consumption of each part. We first tested 101 blocks carrying 15,129 transactions (S_3 in Table 2) to measure the time consumption of each phase in Vegeta. Figure 8 depicts the results. In this measurement, the speculation phase was based on the latest Ethereum world state, i.e., block B_h is speculatively executed based on the world state of block B_{h-1}. The speculation yields 1.47% re-execution rate. The cost of re-execution is insignificant compared to other parts. The speculation phase consumed more time than the replay phase because of the overhead of ordering transactions and building the DAG for capturing dependencies. It can also be observed that Vegeta spent much less time in executing all phases compared to serial execution. Note that in real systems the speculation phase is executed only by a single node and can be further overlapped with the replay phase of some previous blocks.

Re-execution rate. We conducted evaluation in the re-execution rate by performing the speculation phase based on different world states. We then counted how many transactions are re-executed in the relay phase and accordingly calculated the rate. The results are shown in Figure 9. We performed the similar experiment based on the Ethereum data ranging from 2017 to 2023. X-axis indicates the gap of block heights between the speculation and the world state it is based on. For instance, 0 means the speculation results of block B_h is based on the world state of block B_{h-1}, while 9 means the speculation results of block B_h is based on the world state of B_{h-10}. With more recent world states the speculation can obtain more accurate dependencies among transactions, thus reducing the re-execution rate. Even with the most recent state, the re-execution rate is still a non-zero value. This is because in the speculation phase Vegeta executes transactions in a fully parallel manner. A concrete example is actually depicted in Figure 1, where the check-balance operation (step 2) for APEFI token failed during speculation but succeeded in the relay phase, because another transaction (within the same block) transferred a sufficient number of tokens to the user's address. The re-execution rate is relatively small except 2021, presumably due to the explosion of DeFi and NFT applications.

Speedup. We compared Vegeta with AriaFB and serial execution to demonstrate the effectiveness of our approach. In the speculate-order-replay framework, the burden of speculation can be evenly distributed to every node and amortized by pipelining, while the replay (and re-execution) phase must be performed by every node. Therefore, in the single-node test, we only measured the time it takes to perform the replay phase and compared it with that of other approaches. The speedup can be considered the upper bound Vegeta can achieve for the given Ethereum workloads.

The evaluation was conducted on the datasets S_1 and S_4 (in Table 2), each of which contains 5,000 blocks. The results are depicted in Figure 10. The performance bottleneck of Vegeta for Ethereum datasets mainly stems from the longest dependency chain, which was also discussed in [35]. Transactions in the same dependency chain must be executed in serial order. We also observe that the longest dependency chain is generated by Wrapped Ethereum (WETH) decentralized application [10], which swaps ethers (cryptocurrency in Ethereum) between layer-1 (native coin) and layer-2 (token) for facilitating trade with ERC-20 tokens. The ratio shown in Table 2 also reflects an upper limit to some extent (8.39× versus 7.8× and 7.7×).

AriaFB treated each block as a batch and first executed all transactions in parallel. If any transaction had a forward dependency (according to Rule 2), the transaction was aborted. We found that without fallback, aborted transactions in vanilla Aria kept accumulating and eventually led to out-of-memory error. We thus integrated our replay phase into vanilla Aria

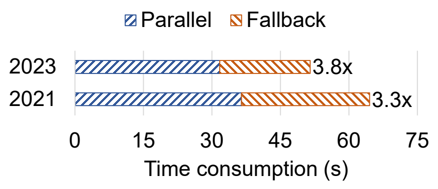


Figure 11: Time consumption and speedups of AriaFB.

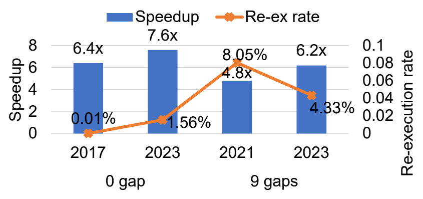


Figure 12: Speedups with different re-execution rates.

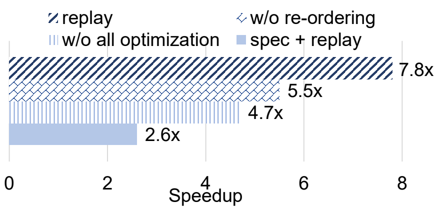


Figure 13: Ablation study of Vegeta.

and obtained AriaFB[2]. For datasets S_1 and S_4, respectively, 40.2% and 42.3% transactions were aborted and further resorted to the fallback for re-execution. Compared to AriaFB, the speedup gain of Vegeta largely comes from the speculation that wisely orders transactions and groups them into batches, effectively reducing abort (re-execution) rate. We further measured the execution time of each part under the datasets of 2021 and 2023 (as shown in Figure 11). The abort rate of 2023 dataset is 40.2%, and the corresponding speedup is 3.8×, while the abort rate of 2021 dataset is 52.1%, so the speedup decreased to 3.3×.

To assess the impact of re-execution rates, we also measured the speedups with the datasets from different years (thus leading to different re-execution rates), as shown in Figure 12. The datasets from 2017 and 2023 exhibit lower re-execution rates, resulting in higher speedups. However, the longest dependency chain problem in the 2017 dataset has a greater impact on the degree of parallelism, leading to a lower speedup compared to the 2023 dataset. To evaluate the performance with even higher re-execution rates, we then purposely performed the speculation phase with a gap of 9 blocks. The speedups were negatively affected.

Ablation study of Vegeta. We evaluated the influence of each optimization in Vegeta. The results are shown in Figure 13. To assess the impact of re-ordering in the speculation phase, we purposely disabled this approach (i.e., sorting and selecting dependency chains, Line 4-9 in Algorithm 1) and preserved the original order in each Ethereum block. The speedup (without re-ordering) decreased to 5.5×, as transactions in some longer dependency chains may start execution later during the relay phase, highlighting that transaction order significantly impacts performance improvement. We emphasize that in real-world deployment, there is no concept of re-ordering, as proposers can freely choose in what order transactions are executed. We then stepped further and removed the optimization for re-execution, which is described in Algorithm 3. The speedup decreased to 4.7×. Finally, we evaluated the performance of Vegeta when both phases (i.e., speculation and replay) are performed after consensus by every node. It can be regarded as a deterministic concurrency control protocol based on the order-execute framework. The speedup decreased to 2.6×. This result demonstrates the benefit of distributing speculation across all nodes.

[2]As also discussed in [56], any deterministic concurrency control can potentially be used.

Table 3: Re-execution rates in multi-node test.

	S_2	S_4
4 nodes	1.66%	1.58%
10 nodes	1.89%	1.82%

Multi-core scalability. To evaluate the multi-core scalability, we conducted experiment with different configurations of 2, 4, 8, 16 and 32 cores. The results are shown in Figure 14. The speedup of Vegeta grows linearly from 2 to 8 cores. As the number of cores continues to increase, this trend diminishes.

6.3 Multi-Node Performance

When it came to multi-node deployment, we integrated transaction execution with a consensus module. We set the number of nodes n to 4 and 10 in this experiment, which respectively tolerates 1 and 3 Byzantine nodes. We did not conduct even larger-scale experiments because the Ethereum workloads were generated sequentially. With larger n, there will be a larger gap between speculation and replay, which are not realistic in practice. Nonetheless, as Vegeta aims to provide scalability in the speculation phase, we expect Vegeta will perform even better when n becomes larger. The evaluation was conducted on the datasets S_2 and S_4 in Table 2.

As with a leaderless consensus protocol, we evenly assigned blocks to every node, i.e., node i is responsible for speculatively executing block B_j (in Vegeta) and proposing B_j in consensus if $j\%n = i$. When the consensus layer outputs, n blocks are delivered to the upper layer for execution. These n blocks are totally ordered based on proposers' IDs.

Re-execution rate. As already shown in Figure 9, a more recent world state can provide a higher precision for dependency analysis. Due to this reason, a node would not perform speculation of a new block until its remaining blocks ready for replay is equal to or less than K. We set K to 2 in this experiment because as shown in Figure 8, the time taken by speculation is less than twice the replay time. Table 3 shows the re-execution rate.

Leaderless vs. leader-based protocols. We first compared the performance of Vegeta integrated with BKR (a leaderless protocol) with that integrated with a leader-based protocol.

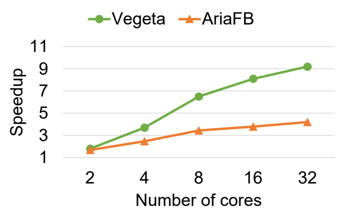


Figure 14: multi-core scalability.

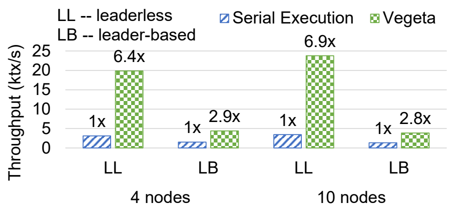


Figure 15: Throughput of leaderless and leader-based protocols.

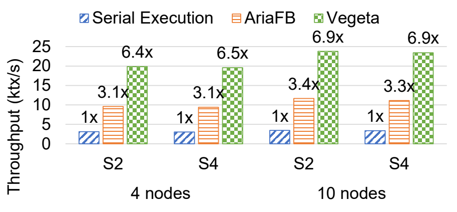


Figure 16: Throughput and speedups in multi-node test.

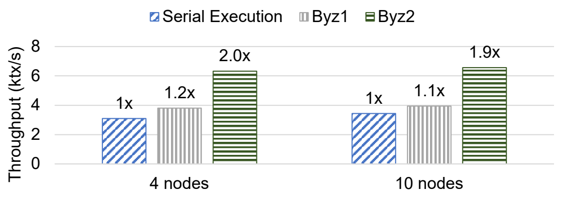


Figure 17: Throughput under Byzantine faults.

For the latter case, our evaluation chose PBFT [25] as the ordering module. The single leader performs the speculation phase, while every node replays the block after consensus. The evaluation was conducted on the datasets S_2 in Table 2, and the results are shown in Figure 15. Because the single leader took the heaviest burden of computation, the leader node throttled Vegeta's performance.

Speedup. The results are shown in Figure 16. Being affected by consensus, the throughput of all protocols decreased compared to Figure 10. Because of the competition between speculation and replay for computing resources, the performance of Vegeta is further affected. Although for 10-node case the re-execution rate is slightly higher than that for 4-node case (Table 3), the $n = 10$ throughput raised because with larger n, leaderless consensus can commit more proposals in each instance. The performance of serial execution and AriaFB exhibited a similar trend due to the same reason.

6.4 Performance under Byzantine Faults

As blockchain systems target Byzantine faults, in this experiment we evaluate their impact on performance by configuring f nodes to exhibit two typical Byzantine behaviors. Specifically in our design, a Byzantine node can purposely generate incorrect dependencies and/or read/write sets in the speculation phase. Such an anomaly cannot affect the safety property of Vegeta but may badly affect its performance.

The results are shown in Figure 17. In the first scenario (Byz1), f Byzantine nodes provided a serial order and empty read/write sets for each block proposal, which forced other correct nodes to sequentially execute each transaction twice (one for replay, and the other for re-execution). Note that the proposals of other correct nodes were still correctly speculated. The performance gains coming from correct nodes are almost eliminated by Byzantine nodes. Although in 10-node case Vegeta had higher speedup compared to 4-node case (Figure 16), it also suffered more badly than 4-node case due to the higher proportion of Byzantine nodes (1/4 versus 3/10). We argue that the impact of this typical malicious behavior may be simply mitigated by, e.g., sequentially executing each transaction only once, as the speculation already provided serial order.

The second scenario (Byz2) is a bit more subtle, where Byzantine nodes provided correct dependencies but empty read/write sets, which still led to 100% re-execution rate for their proposals. The throughput of Vegeta still significantly decreased. For the block proposals made by Byzantine nodes, all the transactions were first executed (efficiently) in the replay phase and then re-executed in serial order. Note that Byzantine behaviors may also badly affect the performance of AriaFB, e.g., by deliberately orchestrating a sequence of transactions that causes frequent aborts.

7 Conclusion

Towards efficient transaction processing in blockchains, we presented the speculate-order-replay framework. Our framework is tailored to leaderless consensus, which brings both challenges for deterministic parallelism and opportunities for boosting performance. Our framework allows each node to independently pre-execute transactions and analyze their dependencies in the speculation phase, in order to provide information that enables efficient parallel execution in the replay phase. We instantiated our framework through Vegeta, a transaction-level concurrency control protocol. Experimental evaluation under Ethereum workload showed promising performance compared to other parallel processing schemes.

Acknowledgments

We are very grateful to our shepherd, Xiaoqi Chen, and the anonymous reviewers for their insightful feedback. This work was supported by the National Natural Science Foundation of China (grant no. 62372293) and the Shanghai Action Plan for Science, Technology and Innovation (grant no. 24BC3201300). Shengyun Liu is the corresponding author.

References

[1] Ape finance. https://ape.fi/swap.

[2] Ethereum example for smart contract innocation. https://etherscan.io/tx/0xf195576ee9915449726ccdad80d389cb6a1853cea685368ea98da2b79e1b57c9.

[3] Ethereum is a dark forest. https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest.

[4] Ethereum virtual machine. https://ethereum.org/en/developers/docs/evm/.

[5] Ethereum whitepaper. https://ethereum.org/en/whitepaper/.

[6] Proposer/block builder separation. https://ethresear.ch/t/proposer-block-builder-separation-friendly-fee-market-designs/9725.

[7] Sui documentation: Validator committee. https://docs.sui.io/guides/operator/validator-committee.

[8] Uniswap. https://uniswap.org/.

[9] Usd coin. https://www.circle.com/en/usdc.

[10] Wrapped ethereum. https://101blockchains.com/wrapped-ethereum/.

[11] Zero-knowledge rollups. https://ethereum.org/en/developers/docs/scaling/zk-rollups/.

[12] Daniel J. Abadi and Jose M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 61(9):78–88, aug 2018.

[13] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, page 23–34, New York, NY, USA, 1995. Association for Computing Machinery.

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

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

[16] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, and Roger Wattenhofer. Fnf-bft: Exploring performance limits of BFT protocols. *CoRR*, abs/2009.02235, 2020.

[17] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc. Technical Report SWIRLDS-TR-2016*, 1, 2016.

[18] C. Basile, Z. Kalbarczyk, and R.K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 17(5):448–465, 2006.

[19] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 173–184, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 183–192, New York, NY, USA, 1994. Association for Computing Machinery.

[21] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[22] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[23] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. Morty: Scaling concurrency control with re-execution. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 687–702, New York, NY, USA, 2023. Association for Computing Machinery.

[24] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[25] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

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

[27] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–10, 2008.

[28] T. Crain, C. Natoli, and V. Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1501–1518, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[29] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

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

[31] Sisi Duan, Michael K. Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2028–2041, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[33] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, jan 2017.

[34] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 15–26, New York, NY, USA, 2014. Association for Computing Machinery.

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

[36] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 232–244, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Neil Giridharan, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Bullshark: DAG BFT protocols made practical. *CoRR*, abs/2201.05677, 2022.

[39] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, 2019.

[40] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, jun 1996.

[41] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 803–818, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[43] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 164–173, 2000.

[44] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering*, 34(11):5070–5083, 2022.

[45] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[46] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 237–250, Berkeley, CA, USA, 2012. USENIX Association.

[47] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584, 2004.

[49] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2007.

[50] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[51] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

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

[53] Haoran Lin, Yajin Zhou, and Lei Wu. Operation-level concurrent transaction execution for blockchains, 2022.

[54] Jian Liu, Peilun Li, Raymond Cheng, N. Asokan, and Dawn Song. Parallel and asynchronous smart contract execution. *IEEE Trans. Parallel Distrib. Syst.*, 33(5):1097–1108, may 2022.

[55] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous bft consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 264–280, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.

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

[58] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[59] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 35–48, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Afonso Oliveira, Henrique Moniz, and Rodrigo Rodrigues. Alea-bft: Practical asynchronous byzantine fault tolerance. *CoRR*, abs/2202.02071, 2022.

[61] Shujian Qian and Ashvin Goel. Massively parallel Multi-Versioned transaction processing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 765–781, Santa Clara, CA, July 2024. USENIX Association.

[62] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 180–194, New York, NY, USA, 2021. Association for Computing Machinery.

[63] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214, 2022.

[64] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2plsf: Two-phase locking with starvation-freedom. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 39–51, New York, NY, USA, 2023. Association for Computing Machinery.

[65] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990.

[66] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 105–122, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: A software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 69–84, New York, NY, USA, 2022. Association for Computing Machinery.

[68] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 17–33, New York, NY, USA, 2022. Association for Computing Machinery.

[69] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1150–1160. VLDB Endowment, 2007.

[70] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[71] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), Sep. 1997.

[72] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1–2):70–80, sep 2010.

[73] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery.

[74] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 59–72, New York, NY, USA, 2007. Association for Computing Machinery.

[75] Bo Wang, Shengyun Liu, He Dong, Xiangzhe Wang, Wenbo Xu, Jingjing Zhang, Ping Zhong, and Yiming Zhang. Bandle: Asynchronous state machine replication made efficient. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 265–280, New York, NY, USA, 2024. Association for Computing Machinery.

[76] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, jul 2021.

[77] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, Renton, WA, April 2022. USENIX Association.

[78] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

[79] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. Rethinking database high availability with rdma networks. *Proc. VLDB Endow.*, 12(11):1637–1650, jul 2019.

[80] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649. USENIX Association, November 2020.

A Correctness proof of Vegeta

We prove that the execution results on each node are equivalent to the same serial order. Since all information (e.g., transactions and their dependencies) leveraged by the replay phase is agreed upon using consensus and each node executes transactions in a batch-by-batch manner, we only need to prove that each batch produces the same execution results. Note that within a batch, transactions are ordered by their indices. We give the following three lemmas.

Lemma 1. *If nodes i and j execute batch b according to the same world state and tx is re-executed on node i, then tx is re-executed on node j.*

Proof. Within b, every transaction is executed based on the same world state and no one updates any key until they all complete execution. If tx is re-executed on node i, then tx must change its read/write sets (Line 8 in Algorithm 2) or read/write a new key $key' \in all\_keys$ (Line 5 in Algorithm 3). Then obviously node j should also re-execute tx because the read/write sets of each transaction are obtained during speculation and agreed upon by consensus. □

With Lemma 1, in the following we only consider the transactions that are not re-executed. We discuss the cases without and with an improved replay strategy, respectively.

Lemma 2. *(With a simple re-execution strategy in Algorithm 2) The execution results of each batch are equivalent to the same serial order on every node.*

Proof. For any two transactions tx and tx' in the same batch, they should not have WAW dependency, nor should they have both WAR and RAW dependencies (Line 34 in Algorithm 1). Without loss of generality, we assume the serial order of transactions in the batch is: $\cdots \to tx' \to \cdots \to tx \to \cdots$. We then discuss three cases:
Case 1: If they are non-conflicting, they are serializable. And we determine their order based on the order of their indices.
Case 2: If tx WAR-depends on tx', then since the result of execution will only be committed after all transactions in the batch finish execution, tx' cannot read the value written by tx, they are serialized as tx' happens before tx, the assumed serial order is satisfied.
Case 3: If tx RAW-depends on tx', their execution is equivalent to tx happens before tx' (i.e., re-ordering), so the serialized order can be changed to $\cdots \to tx \to \cdots \to tx' \to \cdots$. Also note that tx in this case must have no WAR dependency with any other transaction, so no dependency circle will be generated in the same batch.

In all cases, the execution results of all the transactions within each batch are equivalent to a certain serial order. It is also trivial to see that every node proceeds based on the same world state and consensus-provided information. So their execution results should be equivalent, □

Lemma 3. *(With an improved strategy in Algorithm 3) The execution results of each batch are equivalent to the same serial order on every node.*

Proof. With the proof of Lemma 2, we only need to discuss the following three cases.
Case 1: Neither tx nor tx' reads a new key that is (newly) written by some transaction. In this case, both tx and tx' are committed by following the serial order extracted from DAG (Line 14 in Algorithm 3). Even if tx and tx' newly write the same key key', there is no RAW-dependency between them, so their execution results are still equivalent to the serial order.
Case 2: tx reads a new key key' that is also (newly) written by some transaction, while tx' does not. In this case, tx' gets committed before tx (Line 14), while tx is executed again and committed (Line 16). Even if tx' also writes key', tx can read the update of key' if necessary. Anyway, tx' is executed before tx in the serial order.
Case 3: both tx and tx' read some new keys, say key' and key'', that are also written by some transactions. In this case, both tx and tx' are executed (sequentially) again and committed. Without loss of generality, tx is executed before tx' in the serial order. □

With Lemma 1, every node re-executes the same set of transactions sequentially. With Lemma 2, Lemma 3 and every node executes batches one after another, the execution results of all transactions are equivalent to the same serial order.