Accelerating Decentralized Execution of Blockchain Transactions Towards Centralized Performance

Maya Leshkowitz, Oded Wertheim and Ori Rottenstreich
Orbs
www.orbs.com

***Abstract*—Blockchain mechanisms include two main tasks: transaction ordering and transaction execution. Following a paradigm in which execution is separated from ordering we focus on the execution task. Typically blockchain execution is performed sequentially due to possible dependences between transactions. To allow scalability, we suggest a decentralized execution model in which network nodes interact with a stronger accelerator. The accelerator executes a block of transactions and provides hints that allow committees of nodes to execute in parallel the different block segments. The committees then verify the execution of the accelerator and the validity of the block partition to segments. The protocol guarantees correctness of the execution as well as liveness, even when the accelerator or subset of the nodes are Byzantine. We expedite the execution process towards the performance of a centralized system while minimizing the amount of communication between the players. We evaluate our solution and compare it to existing schemes.**

I. Introduction

A. Transaction ordering and execution

Blockchain mechanisms are composed of two main tasks, *transaction ordering* and *transaction execution*. In the ordering process, nodes reach consensus regarding the ordering of transactions. Transactions can either be simple payments between one node to another or may encompass more general functionality such as of Ethereum smart contracts [17]. Transactions are ordered within a block, and blocks are appended to the chain, implying a full ordering on the transactions. The task of transaction execution is essentially computing the new state of the system which is the outcome of executing the transactions in the block on the previous state. The state contains data such as the balance of every user account. Transaction execution is also responsible for outputting the outcome of execution, also known as transaction receipts.

Classical blockchain implementations couple between these two tasks by requiring execution of the new block to be performed by the block proposer. For example, this is the case in Ethereum [17] where the new state is computed by a miner as part of the newly proposed block. Nodes that receive a block proposal accept it only after re-executing and checking that the state was computed correctly. The coupling between the ordering and execution tasks may undermine efficiency, as these two processes necessitate different resources in terms of storage, bandwidth and computation power. As a result, they admit to different approaches for distributing and scaling.

Parallel transactional execution is not straightforward and can only be done to a limited extent due to dependency between transactions. The problem is even more challenging in execution of Turing complete code, where the state variables accessed during the execution may depend on the current state, thus cannot be determined prior to the execution. A common method for parallel execution is transactional memory [12]. With transactional memory, transactions may be speculatively executed in parallel based on the state prior to the execution. This approach requires roll back in case of conflict, which can be efficiently performed when run by a single party but has high latency overhead in decentralized systems.

B. Our Contribution

Our main result is a paradigm that refers to a blockchain protocol that achieves consensus on the ordering of transactions prior to execution. Our paradigm modifies the way transaction execution is performed, achieving *accelerated transaction execution* in the presence of a strong computational entity, which we refer to as an *accelerator*. The accelerator is a high performance entity (in terms of compute, storage and network capabilities) that is likely to operate as a distributed system over multiple cores and servers enabling it to perform fast execution of the entire block. The accelerator performs the entire execution and supplies executors with hints that allows them to execute and verify partial segments of the block in parallel such that verification of the different segments imply valid execution of the block.

The main advantage of the accelerated protocol is that when the accelerator is honest and the number of executors in the network is moderate, the total running time of the execution is close to that of an efficient execution in a centralized system. Moreover, when the accelerator or some of the executors are faulty, the protocol still enjoys liveness and security in execution like a decentralized system. Another significant advantage is that unlike many novel sharding architectures, the parallel execution is performed seamlessly to the application.

As illustrated in Fig. 1, the accelerated transaction execution protocol works by breaking the transaction execution of a block into disjoint segments of consecutive transactions. The accelerator performs the execution of the entire block and while doing so saves for each segment $i \in [1, n]$ the write operations that are the result of execution of the first i segments. Other network nodes are organized in committees and serve as executors. Committee i is in charge of verifying the execution of segment i, using the write operations of the first $i - 1$ segments as input for execution. If the transaction

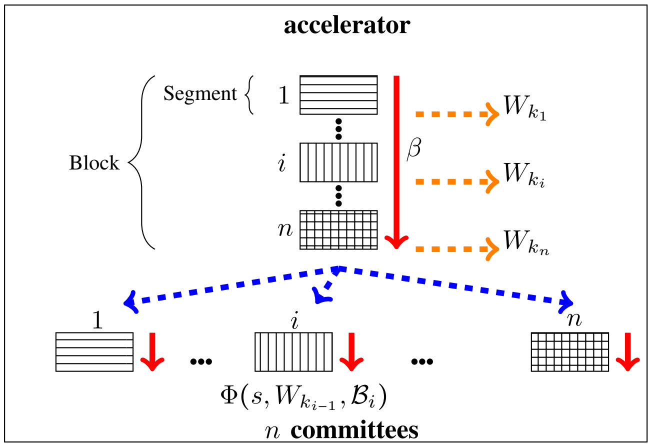

Fig. 1. Illustration of a block transaction execution. The accelerator performs execution β times faster than a (regular) executor. The accelerator provides committee $i \in [1, n]$ hints (like $W_{k_{i-1}}$) needed for executing the i^{th} segment $\mathcal{B}_i$ and for checking consistency between segments. This allows parallel execution of the different segments by the committees and checking that the block was executed correctly by the accelerator. Each committee is composed of several executors, each performing this check.

execution by the accelerator was not performed correctly then at least one segment was not executed correctly or there is a pair of adjacent segments that are not compatible. In both cases at least one committee would detect this and notify the other network nodes. The protocol then falls to the base execution protocol where execution is performed independently by the executors until the faulty accelerator is replaced.

The protocol incorporates several techniques to overcome inherent challenges for achieving correctness, liveness and communication efficiency when both the accelerator and a certain fraction of the executors may be faulty. In order to achieve correctness we need the executors to ensure compatibility between the execution of different segments. We do this by having the executor send an execution digest, which is a succinct message containing necessary information about the execution of all the segments. When all committees approve the execution digest it follows that the entire execution was performed correctly.

The executors are also required to validate the partition of the transactions into segments, which is performed by the accelerator. Transactions should not be edited and their order should be maintained. This is especially crucial as well as challenging when committees receive only the transactions of their segment and not all transactions. We design a proof technique that allows the accelerator to prove the partition validity through its approval by all committees.

II. Related work

The common art in most of the blockchain decentralized architectures is to have all the nodes validate by executing every block of transactions. In such architectures, increasing the network size does not increase the network capacity and the network scale is limited.

An approach for addressing blockchain scalability and in particular tokens or asset management applications is by L2 network architecture, such as State channels [3] or Plasma [2]. L2 networks are built on top of a main blockchain and increase scalability by transitioning some of the state handling off-chain while relying on the main chain for synchronization, security and dispute mediation. Hyperledger Fabric [5] introduces an execution-ordering-validation paradigm performing execution based on the current state. Transactions are ordered after execution and then validated, ensuring that the execution is consistent with current state. Transactions executed on a stale state are marked as invalid and their effects are disregarded.

Other architectures such as Telegram Open Network [11] and Polkadot [1] implement a sharding scheme to address network scalability. In these sharding schemes, the network state, users and participating nodes are divided into shards, allowing each shard to operate independently. Cross-shard operations are performed by a messaging scheme, a transaction is first executed on the sender shard, as a result one or more messages along with an execution proof may be sent to other shards to continue the execution. While these sharding schemes address the network scale, they are not transparent to a general purpose application and in particular require special handling of atomic cross-shards operations. Methods for increasing the network transaction rate, using software transactional memory methods by the leader node and sending scheduling hints to the validation nodes have also been examined [10].

III. Preliminaries

We overview cryptographic primitives and data structures that we make use of in our protocol.

A. Cryptographic Hash Functions

Along the paper we assume the hash functions used are second preimage resistant. A hash function H is second preimage resistant if it is hard to find for a given m_1 a preimage $m_2 \neq m_1$ such that $H(m_1) = H(m_2)$. We denote by L the length in bits of the hash value.

B. Merkle Tree

A Merkle tree is a known tool in cryptography, first suggested by Merkle [13] which enables proving a membership of a data element in a set efficiently, without revealing the entire set. In a Merkle tree, every node has a *Merkle label*. For the leaves this label is the hash of a data block, and for every non leaf node this label is the hash of the concatenation of the labels of its children (or the label of its child in case it only has one child). In order to verify that some data is included in a Merkle tree T, one needs to obtain from a trusted source the label of the Merkle root of the tree, which we denote by $M(T)$. A Merkle proof for the containment of some data v, which corresponds to a leaf in the tree, consists of the sibling path of the leaf, which contains the labels of all the siblings of the nodes in a path from the leaf to the root. We assume Merkle trees are second preimage resistant, making it impossible to reproduce a Merkle root label.

C. Threshold Cryptography

Threshold cryptography [7], [9], [16] refers broadly to techniques for allowing *joint groups* of entities to use a cryptographic system, be it to compute signatures, or to decrypt data. In the context of this work, we make use of threshold signatures. In particular, a (t,c)-threshold signature scheme is executed by c entities, any t of which (for some fixed $t \in [2,c]$) are able to sign a message successfully. Threshold security guarantees that whenever a specified hardness assumption holds, any attempt by up to $t-1$ of the entities to sign a message is bound to fail.

IV. Model and Definitions

A. Players and Byzantine model

We assume that the identities of the accelerator and all executors are known in advance. We assume that the executors are partitioned to n groups called *committees* which are static[1]. For simplicity of presentation we assume all committees have the same number of members c. Each participant is aware of the identities of executors in its committee, and is aware of identifiable data regarding other committees (such as the public key associated with each committee).

We regard nodes that are either sleepy or dishonest as faulty. We assume that there are at most α fraction of faulty executors in each committee (where $\alpha \cdot c$ is an integer), and $\alpha < \frac{1}{2}$ is a known system parameter. When assembling the committees there is need to make sure that this assumption is justified by taking large enough committees based on α.

For simplicity, we assume that every executor that is non-faulty (i.e., is alive and honest) has the same computational power. It suffices to assume that each executor has at least a certain ammount of computational power. Denote by $\beta > 1$ the acceleration parameter, which means that the accelerator performs computations β times faster than the executors.

Messages are either destined to a committee or to all executors and we assume nodes propagate every message they have not received before to all other nodes the message is destined to. Signatures are used to authenticate the sender's identity. Throughout our work, we assume a strong synchronous network. This means that there is a known fixed bound δ, such that every message delays at most δ time when sent from one point in the network to another.

B. Ordering Service

We assume the availability of an external transaction ordering service. The ordering service is responsible of ordering transactions in blocks and reaching consensus on the ordering of these blocks. Transactions in a block are organized in a Merkle tree such that each transaction corresponds to a leaf. The Merkle root of the tree is contained in the block header. We assume that every transaction block received from the ordering service is final in the sense that once a new block of some height is received it will never be replaced by a different block. In our setting the ordering service is not responsible for performing validations that require executing the transaction and is not responsible for computing the final state of the block. Examples for consensus algorithms suitable for ordering services include [4], [6], [8], [14].

While acceleration of the execution by parallel computation is complex due to the sequential nature of the execution and dependency between transactions, parallel ordering is relatively simpler and can be achieved by techniques such as hierarchical consensus [18].

C. State transition

We refer to the state as a map data structure containing memory addresses and the values stored in these addresses which enables read and write operations. Transactions are commands that include computations and changes to be performed to the state (this can include both simple and complex computations). State transition is a deterministic process in which ordered transactions in a block are executed sequentially, receipts and write operations are created and then applied to the current state. We divide the description of state transition to two processes, *execution* and *applying write operations*. We provide details on each of them below.

Transaction execution: Transaction execution is a process in which the validity of each transaction is checked, computations are carried out, and the write operations to the state that should be performed as a result are determined. Write operations are maintained in the format of (memory address, value) in an *aggregated writes* data structure, where we denote by W_j the aggregated writes that contains the write operations of transactions $tx_1, \dots, tx_j$. An example for such data structure is a Cuckoo table [15] which enables read operation within a constant time and efficient insertion. Execution also outputs a receipt for each transaction, which contains the outcome of the transaction's execution (such as outputs, failure notices, etc.).

The input to the execution function ϕ is a tuple (s, W_j, tx_{j+1}) where s is the state, tx_{j+1} is the transaction to be executed and W_j is the aggregated writes up to the j^{th} transaction $tx_1, \dots, tx_j$. These are transactions that have already been executed but have not been applied to the state yet. Executing tx_{j+1} may include complex computations and reading and writing multiple values from the memory. The state s and the aggregated write operations together serve as the input memory for execution. When the execution of tx_{j+1} requires writing some value v to memory address m' then a tuple (m', v) is inserted to the aggregated writes. The output of transaction execution is a tuple (W_{j+1}, r_{j+1}), where W_{j+1} is the new aggregated writes and r_{j+1} is the receipt.

We denote by Φ the execution function for a transaction list (not necessarily a block), $\mathcal{B} = (tx_{j+1}, \dots, tx_{j+b})$. The input to Φ is a tuple $(s, W_j, \mathcal{B})$, where s and W_j are again the state and aggregated writes. The output is a tuple $(W_{j+b}, R^{\mathcal{B}})$ which contains the aggregated writes and the receipts of all transactions in $\mathcal{B}$. The execution function Φ executes each transaction sequentially using as input the aggregated writes of all previous transactions in $\mathcal{B}$. That is, the function $\Phi(s, W_j, \mathcal{B})$ performs for $i = 1, \dots, b$: $(W_{j+i}, r_{j+i}) = \phi(s, W_{j+i-1}, tx_{j+i})$. The function outputs the aggregated writes $W^{\mathcal{B}} := W_{j+b}$ and receipts $R^{\mathcal{B}} := (r_1, \dots, r_b)$.

[1]Removing the assumption of static committees can be done by performing periodic key generation for threshold signatures.

Bound on execution time: We bound the compute resources for executing a transaction (such as done in [17]) and if execution exceeds these resources then no write operations are created for it. Using the bound on the compute resources we bound the time complexity of executing a transaction by an executor by a fixed bound T^E_ϕ.

Applying write operations: After executing a block of transactions, the apply write function updates the current state. The apply write function Ψ receives an initial state s and aggregated writes W and updates the state $s \leftarrow \Psi(s, W)$ by adding all the (memory address, value) pairs in W to the state.

D. Execution Service

The execution service is responsible for performing state transition for a transaction block $\mathcal{B} = (tx_1, \dots, tx_b)$ received from the ordering service, and for reaching an agreement on the state transition outputs. The output of state transition of a block is an updated state s, block receipts $R^{\mathcal{B}}$ and aggregated block writes $W^{\mathcal{B}}$. The state is stored in a Merkle tree, indexed by memory addresses, and updated with every block executed[2]. The receipts are stored in a Merkle tree, created for each block of transactions, indexed according to the transaction's number in the block. The block receipts tree $R^{\mathcal{B}}$ is never updated after its creation. The aggregated writes data structure keeps tuples of memory addresses and their value allowing efficient read and write operations.

For every block $\mathcal{B}$ received from the ordering service an execution digest is created by the execution service which includes:

- The block height number ℓ.
- A hash pointer of the ℓ^{th} ordering block $\mathcal{B}$ header.
- The Merkle root of the state tree after the execution of block $\ell - 1$.
- The Merkle root of the block receipts tree $R^{\mathcal{B}}$.
- The hash of the aggregated block writes, $W^{\mathcal{B}}$.

For each executed block $\mathcal{B}$, the execution service reaches agreement on the outputs s, $W^{\mathcal{B}}$ and $R^{\mathcal{B}}$, and a certificate for the execution digest is created in the process. Reaching agreement on transaction execution is important for maintaining a reliable record which enables updating other network participants (such as other executors, the ordering service and the users) on the outcome of the execution process.

V. The Partition Proof

In the protocol the accelerator partitions an (ordered) block $\mathcal{B}$ into n disjoint *block segments* $\mathcal{B}^1, \dots, \mathcal{B}^n$ and sends segment $\mathcal{B}^i$ to the i^{th} committee. We would like to make sure that the partition is valid such that the transactions in $\mathcal{B}^1, \dots, \mathcal{B}^n$ are identical to those of $\mathcal{B} = (tx_1, \dots, tx_b)$ and they appear in the same order. Existing transactions should not be modified or omitted and new transactions should not be added.

[2]Storing the state in a Merkle tree enables maintaining a hash fingerprint (the value of the Merkle root of the state tree) for the entire state which can be efficiently updated when applying write operations for the block. Using a Merkle tree for the state also enables supplying efficient proofs of values stored in the state to network participants that are not executors.

The accelerator demonstrates this through a *partition proof* μ that it sends to all n committees. Each committee $i \in [1, n]$ verifies μ with the help of information taken from its segment $\mathcal{B}^i$ and signs the proof μ if the test passes. The design of μ has the guarantee that μ is approved by all non-faulty executors if and only if the partition is valid. Furthermore, for communication efficiency the proof should be short. An inherent challenge for such verification process is the partial view each committee has based on the transactions it receives, while the correctness of the partition is affected from the relation between block segments (such as disjointness and full coverage of the block).

Construction of a partition proof. We refer to the Merkle tree of the block of transactions $\mathcal{B}$ as T. In the Merkle tree, transactions appear as leaves and an internal node's hash value is computed based on the hash values of its direct descendants. Executors receive the Merkle root for T from the ordering service in a secure way, as part of the block header. Consider the set of allocated transactions $\mathcal{B}^i$ for a committee $i \in [1, n]$ and the corresponding set of leaves. The block segment $\mathcal{B}^i$ contains the transactions' indices in the block. Segment $\mathcal{B}^i$ is described as a disjoint union of transaction sets, such that each set corresponds to a subtree of the Merkle tree. (By subtree we refer to a node and all of its descendants in the tree.) We consider such a union where each subtree is of a maximal size, namely no two subtrees for $\mathcal{B}^i$ can be merged to a larger subtree. Let $T_i = \{T_{i,1}, T_{i,2}, \dots\}$ be the subtrees for segment $\mathcal{B}^i$ assigned to committee i. Likewise, let $P_i = \{p_{i,1}, p_{i,2}, \dots\}$ be the corresponding Merkle roots for these subtrees. Each root for a subtree is associated with the location of the subtree in the Merkle tree. Denote by ℓ_i the number of transactions in segment $\mathcal{B}^i$. Let $\mu = \{(P_1, \ell_i), \dots, (P_n, \ell_i)\}$ be the partition proof, which includes list of subtree roots for each segment.

Verifying a partition proof. The executor first computes the location of the subtrees in the Merkle tree for every segment using the number of transactions $\ell_1, \dots, \ell_n$. To verify a partition based on its proof and a block segment $\mathcal{B}^i$, an executor in the i^{th} committee performs the following checks:

(C1) Validity of Merkle roots for segment $\mathcal{B}^i$. The executor partitions the transactions of segment $\mathcal{B}^i$ into subtrees using their indices in the block which determine their location in T. It computes the Merkle roots for each subtree and compares them to those in P_i.

(C2) Validity of the Merkle root for the block $\mathcal{B}$. An executor makes use of the Merkle root sets $P_1, \dots, P_n$ along with their locations to compute hash values for larger subtrees composed of transactions from multiple segments. This process is performed bottom-up, starting from the lowest level roots in μ until computing the root of the complete Merkle tree, and comparing it to the value received in the header of block $\mathcal{B}$.

Correctness of proof verification. We show the following property of the proof verification.

Theorem 1: The proof verification correctly determines the validity of a partition in the sense that: *(i)* a valid partition is approved by all non-faulty executors; *(ii)* approval by non-faulty executors from *all* committees implies validity.

Proof outline. We first show that *(i)* holds. Within each segment an executor partitions a segment to subtrees, following the segment's location in the Merkle tree. It then computes the hash values of each of the subtrees based on the transactions in the segment $\mathcal{B}^i$, as done in μ, and hence check *(C1)* passes. Likewise, computing the Merkle root for the complete block $\mathcal{B}$ can be done by the executor based on the hash values of $P_1, \ldots, P_n$. Since the partition is valid, the root computed by the executor matches that in the block header and *(C2)* passes.

We turn to show *(ii)*, explaining that when the set of transactions is modified by the accelerator the partition is not approved by all non-faulty executors of some committee. Executors in each committee compute the Merkle root for the block based on $P_1, \ldots, P_n$. In case of an illegal partition, the sequence of transactions $(\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^n)$ differs from that of the block $\mathcal{B}$. Let i be the first committee for which its assigned segment $\mathcal{B}^i$ implies a change in the sequence. We derive that at least one of the checks *(C1)*, *(C2)* fails for executors in the i^{th} committee and the partition is not verified.

Length of a partition proof. By deriving an upper bound on the number of subtrees that appears in μ, we can deduce the length of the partition proof is at most $(2n \cdot \log_2(b_{max})) \cdot L$. The calculation is omitted due to space constraints.

VI. Accelerated Execution Protocol

The *Accelerated transaction execution protocol* is an interactive protocol between an accelerator[3]and n committees of executors each in charge of executing a segment and verifying that the execution process was performed correctly by the accelerator. In the absence of faulty nodes a single executor would be enough for performing the role of the entire committee. However, since some of the executors may be faulty there is need for multiple executors in a committee in order to ensure that at most an α fraction of the executors in each committee are faulty. The protocol initializes its setup once and then operates in terms such that in each term state transition of a single block is performed. A term is divided to the accelerator's part and the executor's part.

Initial protocol setup. For $i = [1, n]$ the executors in committee i run a distributed key generation protocol of a $(\alpha \cdot c + 1, c)$-threshold signature scheme. The committee public key PK_i is sent to all executors in the other committees. The accelerator generates a secret key used for signing messages.

Term description - the accelerator's part:

Initialization of accelerator's term. Once the previous term has ended and a block $\mathcal{B}$ that succeeds the block from the previous term is available, the accelerator initiates its term.

Partitioning and sending block segments. Given the set of transactions in a block $\mathcal{B} = (tx_1, \ldots, tx_b)$ and the number of committees n, the accelerator runs a process for partitioning the transactions in the block into n (disjoint) consecutive *block segments* $\mathcal{B}^1, \ldots, \mathcal{B}^n$ such that for $i \in [1, n]$ the transactions in the i^{th} segment are $\mathcal{B}^i = (tx_{k_{i-1}+1}, \ldots, tx_{k_i})$ where $k_0 = 0 \le k_1 \le \cdots \le k_n = b$ are the partition indices. Namely, the set of transactions and their order are not modified relative to the original block. The block is partitioned into segments such that computing state transition of each segment takes a similar time, up to some factor $\gamma > 1$.[4]

The accelerator sends executors in committee i the transaction segment $\mathcal{B}^i$ and a *partition proof* μ which serves as a proof that the block was partitioned as required even though the executor does not receive the entire block. We elaborate more on μ and its verification in Section V.

Performing execution and sending execution outputs. The accelerator performs transaction execution of the segments $\mathcal{B}^1, \ldots, \mathcal{B}^n$ in sequential order. That is, for $i \in [1, n]$ it computes $(W_{k_i}, (r_{k_{i-1}+1}, \ldots, r_{k_i})) = \Phi(s, W_{k_{i-1}}, \mathcal{B}^i)$. It then sends each executor the block receipts $R^{\mathcal{B}} = (r_1, \ldots, r_b)$, the block write operations $W^{\mathcal{B}} = W_b$ and the aggregated writes at the end of the $(i-1)^{\text{th}}$ segment, $W_{k_{i-1}}$.

Creating an execution digest $D^{\mathcal{B}}$. The accelerator sends all committees a digest that contains the following:

- The term number.
- The partition proof μ (see Section V).
- The hash values of the aggregated writes by the end of each segment: $H(W_{k_1}), H(W_{k_2}), \ldots, H(W_b = W^{\mathcal{B}})$.
- The Merkle root $M(R^{\mathcal{B}})$ of the block receipts tree.
- The Merkle root of the state s (prior to the block execution).

Applying write operations to state. The accelerator applies the write operations and updates the state: $s \leftarrow \Psi(s, W^{\mathcal{B}})$. This ends the accelerator's part and it can progress to the next term.

Term description - the executor's part:

Initialization of executor's term. Once the previous term has ended and the header for the block $\mathcal{B}$ that succeeds the previous block is available then the executor initiates the term and resets the term clock.

Segment execution and validations. An executor in the i^{th} committee receives $\mathcal{B}^i$, $W_{k_{i-1}}$, $W^{\mathcal{B}}$, $R^{\mathcal{B}}$ and $D^{\mathcal{B}}$ from the accelerator and performs the following checks:

(i) Checking the state Merkle root in the execution digest: The Merkle root of the state in the execution digest should equal to the Merkle root of the state that it holds.

(ii) Validating the partition proof μ: After receiving the partition proof μ and the block segment $\mathcal{B}^i$ the executor checks that $\mathcal{B}^i$ is indeed a valid i^{th} block segment of $\mathcal{B}$ using μ and the transaction Merkle root in the header of $\mathcal{B}$ (Section V).

(iii) Checking consistency between $W_{k_{i-1}}$, $W^{\mathcal{B}}$, $R^{\mathcal{B}}$ and the execution digest: The executor computes the hashes of $W_{k_{i-1}}$, $W^{\mathcal{B}}$ and the value of the Merkle root of $R^{\mathcal{B}}$ and checks that they are equal to the values in the execution digest $D^{\mathcal{B}}$.

(iv) Executing the i^{th} segment: The executor executes the i^{th} segment by computing: $(W_{k_i}, (r_{k_{i-1}+1}, \ldots, r_{k_i})) = \Phi(s, W_{k_{i-1}}, \mathcal{B}^i)$ and checks that the hash of W_{k_i} is equal to the value in the execution digest, and that the receipts $(r_{k_{i-1}+1}, \ldots, r_{k_i})$ are equal to the receipts of transactions $k_{i-1} + 1, \ldots, k_i$ in $R^{\mathcal{B}}$.

[3]One might also consider the presence of multiple accelerators such that at a given time at most one of them operates. Then, upon identifying misbehavior of the accelerator, the operating accelerator can be replaced by another. For simplicity along the paper we refer to the existence of a single accelerator and whenever the accelerator is faulty the base protocol is employed.

[4]This can be done by relying on time observed in the accelerator execution.

If one of the checks fails or the executor received multiple copies of the same message (for example, multiple versions of the execution digest) then the executor signs a fail message $(\text{Fail}, \text{term\_num})$ for the term and sends it to the other executors in its committee. (See collecting failure signatures below.) Otherwise, if all checks pass, then the executor signs the execution digest and forwards its signature to the other executors in its committee.

Reaching a verdict on a segment within a committee. Within each committee $i \in [1, n]$, the executors collect signature shares for the execution digest. When $\alpha \cdot c + 1$ of the signatures are collected for an execution digest $D^{\mathcal{B}}$, then the executor combines the shares to generate a single committee signature $Sig_i(D^{\mathcal{B}})$ (using the threshold signature scheme), and forwards this signature to the other committees.

Constructing a certificate $Cert(D^{\mathcal{B}})$. The executor collects all the committee signatures in order to create a certificate for the execution $Cert(D^{\mathcal{B}}) = (Sig_1(D^{\mathcal{B}}), \ldots, Sig_n(D^{\mathcal{B}}))$. Otherwise, if the executor did not receive a certificate for the digest before time T^A (defined in the running time analysis, see 'Term time'), and it did not do so already, the executor signs a failure message for the term.

Collecting failure signatures. Throughout the term, the executor collects failure messages from the executors in its committee, and if $\alpha \cdot c + 1$ signatures are received for failing some term ℓ (which might not be the term it is in right now), the executor combines them to generate a committee failure message and forwards this message to the other committees. If the executor receives a committee failure message it falls to base execution protocol, starting execution from the first block it does not have a certificate for.

Applying block write operations. The executor updates the state using the block write operations: $s \leftarrow \Psi(s, W^{\mathcal{B}})$, this ends the term.

VII. Base execution protocol

The base execution protocol is used in the absence of an accelerator or after identifying a problem when running the accelerated execution protocol. The protocol is run by each executor in each committee independently and it proceeds in terms, where in each term a single block is executed and certificate for its execution is generated.

Initial protocol setup. In each committee, the executors run a distributed key generation protocol of a $(\alpha{\cdot}c{+}1, c)$-threshold signature scheme.

Term description:

Term initialization. Once the previous term has ended and the header of the block $\mathcal{B}$ for execution is received, the executor requests the block from the ordering service and initiates its term.

1. **Executing the block.** The executor computes the Merkle root of the transaction block $\mathcal{B}$ that it received and checks that it is equal to the block header. It then executes the block by computing $(W^{\mathcal{B}}, R^{\mathcal{B}}) = \Phi(s, \varnothing, \mathcal{B})$.

2. **Creating and signing the execution digest.** The executor creates an execution digest $D^{\mathcal{B}}$ containing the required values described in section IV-D. The executor then signs the digest and sends the signed digest to its committee members.

3. **Constructing a certificate $Cert(D^{\mathcal{B}})$ for the digest.** The executor collects signatures for the execution digest from $\alpha{\cdot}c{+}1$ committee members and combines them to a committee signature using the threshold signature scheme.

4. **Applying write operations.** The executor applies the write operations $s \leftarrow \Psi(s, W^{\mathcal{B}})$ in order to compute the new state s and proceeds to the next term.

VIII. Running time analysis

We use the following upper bounds on system parameters:

- The time for processing and execution of one transaction by the executors in the base protocol, including the time required for computing the transaction Merkle tree, creating write operations, a receipt and the hash of the aggregated writes is at most T^E_ϕ.
- The number of transactions in a block is at most b_{max}.
- The amount of time it takes an executor to sign, send and verify a peer signature is δ_{sig}.
- A messages sent from the accelerator reaches all relevant non-faulty executors within δ time.
- Checking the aggregated writes for a block takes at most δ_H time for each transaction in the block.
- Computing the Merkle root of the transaction block from the block partition proof takes at most $n \cdot \delta_{\text{part}}$ time.

When analyzing the running time, we neglect the time of applying write operations to the state. The rational behind it is that the state can be partitioned to independent shards so the executor and accelerator can parallelize the associated computations as needed such that the overall time for applying writes is lower than the execution time. Applying write operations can be performed in parallel to executing, because in practice the state transition outputs are available for read operations even before they are applied to the state.

A. Upper bounds on running time - base protocol

Following the above bounds a term of the base execution protocol terminates in $T^B := T^E_\phi \cdot b_{\text{max}} + c \cdot \delta_{\text{sig}} + \delta$ time.

B. Upper bounds on running time - accelerated protocol

Accelerator's part: We assume that the execution time of a transaction by the executors is at most T^E_ϕ. Hence the execution time of a block by the accelerator is at most $\frac{1}{\beta} \cdot b_{\text{max}} \cdot T^E_\phi$. Transaction partitioning and creating a partition proof amounts to computing the Merkle tree of the transaction block, which we count as part of the time of transaction processing and execution. The accelerator also needs to compute hashes of the aggregated writes (of size $O(b_{\text{max}})$) after each segment, as apposed to once for the entire block execution. However, in practice this does not require extra time since once the execution of a segment is completed, the accelerator may start computing the related hash in parallel to the execution of the next segment. It follows that the total bound on the accelerator's part is $T^{AP} := \frac{1}{\beta} \cdot b_{\text{max}} \cdot T^E_\phi$.

Executor's part: The time bound on the executor's part is $T^{EP} := n \cdot \delta_{\text{part}} + \gamma \cdot \frac{b_{\max}}{n} \cdot T_\phi^E + (c+n) \cdot \delta_{\text{sig}} + b_{\max} \cdot \delta_H$, where the details are as follows:

Checking transactions partitioning. This includes two checks. The first, validating the Merkle roots for the segment, is counted as part of the execution time. The second, computing the Merkle root of the block using the partition proof requires at most $2n \cdot \log b$ hash computations (see Section V). We denote the total time this takes by $n \cdot \delta_{\text{part}}$.

Segment i transactions execution. The segments are partitioned so that execution of a segment by the executors takes at most $\gamma \cdot \frac{b_{\max}}{n} \cdot T_\phi^E$ time.

Checking consistency between the aggregated writes and the execution digest. This requires computing hashes of values proportional to the number of transactions in $\mathcal{B}$. We denote by $b_{\max} \cdot \delta_H$ the upper bound of this computation time.

Reaching a verdict on the block. Requires to receive and verify signatures first within the c executors in the committee, and then of the other n committees which in total takes at most $(c+n) \cdot \delta_{\text{sig}}$ time.

Term time T^A: Summing the accelerator's running time, T^{AP}, the time it takes for the accelerator's message to reach the executors, δ, the executor's running time, T^{EP}, and the difference between term clocks, δ, we obtain the total accelerated protocol term time: $T^A := T^{AP} + T^{EP} + 2\delta = \frac{1}{\beta} \cdot b_{\max} \cdot T_\phi^E + n \cdot \delta_{\text{part}} + \gamma \cdot \frac{b_{\max}}{n} \cdot T_\phi^E + (c+n) \cdot \delta_{\text{sig}} + b_{\max} \cdot \delta_H + 2\delta$.

C. Required committee size

Our model assumes that in each committee the ratio of faulty executors is at most $\alpha < 0.5$. We explain how this is achieved with high probability by taking a sufficiently large committee size. Assume the probability of an executor to be faulty is p for some $p < \alpha$, such that this happens to two executors independently. In a committee of size c the probability for exactly k faulty executors is $\binom{c}{k} p^k (1-p)^{c-k}$. Their expected number is $p \cdot c$. By Hoeffding's inequality the probability $P(k \geq \alpha \cdot c) = P(k \geq (p + (\alpha - p)) \cdot c)$ is at most $e^{-2 \cdot (\alpha - p)^2 c}$. If the probability has to be bounded by some small ϵ, a lower bound of $-0.5 \log \epsilon / (\alpha - p)^2$ follows for the committee size c. The lower bound on a committee size implies an upper bound on the number of committees (assuming committees have disjoint sets of executors).

IX. Correctness and Liveness of the protocol

In this section we demonstrate that the protocol satisfies the fundamental desired properties of *correctness* and *liveness*.

Theorem 2: (Correctness) An execution digest of a term that receives a certificate contains the hash values of the state transition outputs (i.e., the hash of $W^{\mathcal{B}}$, the Merkle roots of $R^{\mathcal{B}}$ and the state of the previous term) which were correctly computed using the state transition function and all non-faulty executors hold these outputs.

Proof outline. We prove the claim by induction, assuming that up to some term an execution digest that receives a certificate contains the hash values of the correct state transition outputs, and we show this also holds for the next term. Denote by $D^{\mathcal{B}}$ an execution digest that receives a certificate for the term. In each committee $\alpha \cdot c + 1$ executors signed $D^{\mathcal{B}}$, so at least one non-faulty executor from each committee signed the digest. We denote these executors by $e_1, \ldots, e_n$. From the induction hypothesis $e_1, \ldots, e_n$ also hold the correct state at the beginning of the term.

We distinguish between two cases, according to if the certificate was obtained in the base execution protocol, or the accelerated protocol. If the certificate was obtained using the base protocol, then it is easy to see that it contains the hash values of the correct state transition outputs, since $e_1, \ldots e_n$ are non-faulty. In the second case the certificate was obtained using the accelerated protocol. First note that $e_1, \ldots, e_n$ all received the same aggregated block writes $W^{\mathcal{B}}$ and block receipts $R^{\mathcal{B}}$ from the accelerator, since otherwise their hash and Merkle root would differ from those in $D^{\mathcal{B}}$ and check *(iii)* fails. Since they all signed the digest checks *(iv)* and *(iii)* passed for $e_1, \ldots, e_n$ and hence $W^{\mathcal{B}}$ and $R^{\mathcal{B}}$ were computed correctly.

Theorem 3: (Liveness) For every term, every non-faulty executor obtains a certificate for the execution digest by time $T^A + T^B + 3\delta$ (where T^A,T^B and δ are defined in Section VIII), regardless of whether the accelerator is faulty. Moreover, when the accelerator is non-faulty, the outputs of state transition and the certificate are obtained using the accelerated execution protocol by time T^A.

Proof outline. Summing the time of each process, and since executors initiate their term at most δ time apart, for the accelerated protocol if some non-faulty executor receives a certificate by T^A then all nodes receive it by time $T^A + 2\delta$ (according to their term clock). Otherwise all $(1-\alpha) \cdot c$ non-faulty executors do not obtain a certificate, sign a failure message, collect $\alpha \cdot c + 1$ failure signatures (since $\alpha < 0.5$) and fall to the base protocol. For the base protocol one can readily verify that signatures of the non-faulty executors suffice for generating a certificate for each term within $T^B + \delta$ time.

X. Evaluation

For better understanding the accelerated execution protocol we examine the impact of various system parameters such as the acceleration ratio and the committees number. We also compare the results to an existing approach.

A. Setup

We assume a general purpose application where T_ϕ^E is the dominant factor compared to $\delta_{\text{sig}}, \delta_{\text{part}}$ and δ_H (defined in Section VIII) since it requires state access and non-trivial code execution. Hence, in our analysis we assume $\delta_H \leq T_\phi^E / k_H$, $\delta_{\text{part}} \leq T_\phi^E / k_{\text{part}}$ and $\delta_{\text{sig}} \leq T_\phi^E / k_{\text{sig}}$ where k_H, k_{part} and k_{sig} are fixed constants. These constants may be increased by using a hashing scheme that supports parallel computation, such as Merkle hashing, and by parallelizing signature verification.

Using the above notation, in the base protocol, the total term time is $T^B = T_\phi^E \cdot b_{\max} + c \cdot \frac{T_\phi^E}{k_{\text{sig}}} + \delta = T_\phi^E \cdot b_{\max} \cdot \left(1 + \frac{c}{k_{\text{sig}} \cdot b_{\max}}\right) + \delta$. When the size of $b_{\max}$ is moderately large we can neglect δ as it does not grow with $b_{\max}$ and get that the term time is approximately $T^B \approx T_\phi^E \cdot b_{\max}$.

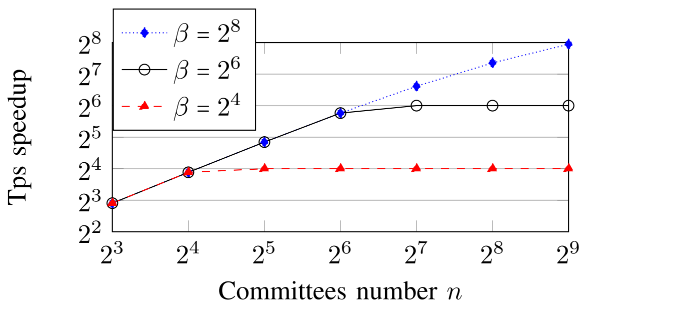

Fig. 2. Impact of the acceleration factor β and the committees number n, $k_H = 500, \gamma = 1.05$.

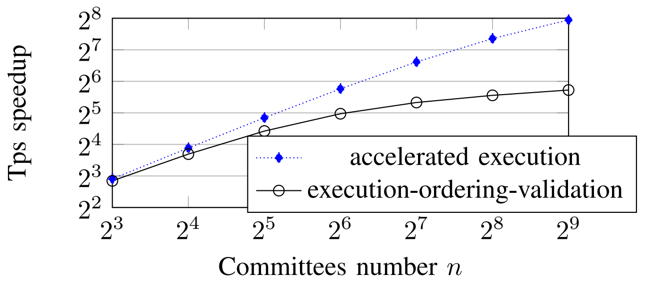

Fig. 3. Accelerated execution vs execution-ordering-validation [5] as a function of the committee number n and the acceleration factor $\beta > n \cdot \gamma$.

We denote Tps^A as the accelerated protocol throughput. As the accelerator starts the execution of the next block once it completes the execution of the current one, the accelerator and executors operate in a pipeline, hence the $Tps^A = b_{max}/max\left(T^{AP}, T^{EP}\right)$. For large b_{max} we neglect the parameters that do not scale with b_{max}, getting $Tps^A \approx min\left(\beta, \frac{n \cdot k_H}{\gamma \cdot k_H + n}\right)/T^E_\phi$.

B. Throughput comparison

In Fig. 2 we compare the throughput of the accelerated protocol, Tps^A, with the throughput of a sequential execution by a single executor $\frac{b_{\max}}{T^B} \approx \frac{1}{T^E_\phi}$. We use $k_H = 500$ based on analysis of the transaction processing and execution time and SHA256 hash computation time of a single state update, as performed by a typical transaction and $\gamma = 1.05$. We note, that increasing n beyond $\beta \cdot \gamma$ does not provide additional speedup.

In Fig. 3 and Fig. 4 we compare the performance of our protocol (with $k_H = 500, \gamma = 1.05$) to an execution-ordering-validation approach (such as Hyperledger Fabric [5], see Section II) with execution performed by n committees of executors. We assume an execution of 5000 transaction per committee before a state update (corresponds to $5sec \cdot 1000Tps$, where 5 seconds is an estimation of the time of execution, ordering and validation). We model the state access distribution as normal distribution $\mathcal{N}(\mu, \sigma)$, namely state index i is accessed by a transaction with probability $\int_i^{i+1} \frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{(x-\mu)^2}{2\sigma^2} dx$. In Fig. 3 we assume $\sigma = 10^6$ and examine the impact of n. As the level of concurrency n increases, the probability of state conflicts resulting in transaction abortion increases, limiting the ability to scale to large n. In Fig. 4 we analyze how the two approaches performance depends on the transaction content.

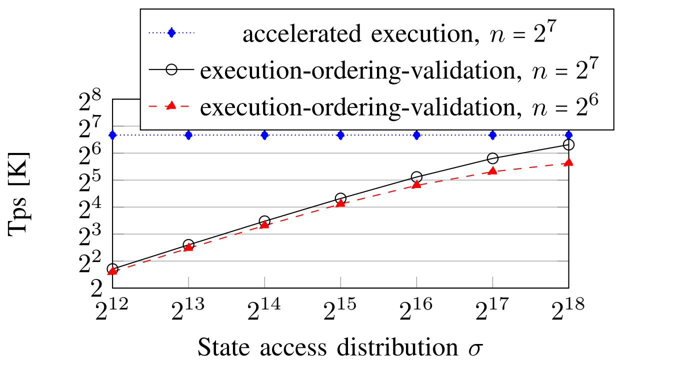

Fig. 4. Accelerated execution vs execution-ordering-validation [5] as a function of the state access distribution.

We set $n = 128$ and evaluate the number of transactions per second (Tps). While the transaction content does not impact the performance of the accelerated execution approach, with execution-ordering-validation, the effective throughput is significantly reduced as the number of collisions increases. We note that for $\sigma < 2^{15}$, doubling the number of parallel committees has a little impact on the speedup.

XI. Conclusion

We introduced an accelerated model for transaction execution in decentralized systems. The model makes use of an accelerator for enabling parallel execution of different block segments by multiple nodes. The protocol relies on proof techniques to guarantee correctness and liveness of the execution even in case of Byzantine behavior. The evaluation shows that the model improves the throughput of existing schemes regardless to the existence of dependencies among transactions. Future work will focus on deriving lower bounds on the amount of protocol communication and allowing committees to maintain a partial state.

References

[1] Polkadot: Vision for a heterogeneous multi-chain framework, 2016.
[2] Plasma: Scalable Autonomous Smart Contracts, 2017.
[3] Sprites and State Channels: Payment Networks that Go Faster than Lightning, 2017.
[4] I. Abraham, G. Gueta, and D. Malkhi. Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR*, abs/1803.05069, 2018.
[5] E. Androulaki et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *EuroSys*, 2018.
[6] A. Asayag et al. A fair consensus protocol for transaction ordering. In *IEEE International Conference on Network Protocols (ICNP)*, 2018.
[7] G. Bleumer. Threshold signature. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1294–1296. 2011.
[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
[9] Y. Desmedt. Threshold cryptography. *Encyclopedia of Cryptography and Security*, pages 1288–1293, 2011.
[10] T. D. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen. Adding concurrency to smart contracts. *Bulletin of the EATCS*, 124, 2018.
[11] N. Durov. Telegram open network. 2017.
[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
[13] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1987.
[14] A. Miller et al. The honey badger of BFT protocols. In *ACM Conference on Computer and Communications Security (SIGSAC)*, 2016.

[15] R. Pagh and F. F. Rodler. Cuckoo hashing. In *European Symposium on Algorithms (ESA)*, 2001.
[16] V. Shoup. Practical threshold signatures. In *EUROCRYPT*, 2000.
[17] G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.
[18] W. Wu, J. Cao, J. Yang, and M. Raynal. A hierarchical consensus protocol for mobile ad hoc networks. In *IEEE International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2006.