Abstract—As the popularity of blockchain technologies has risen, its applications have expanded outside the original cryptocurrency setting. Modern blockchains, such as Ethereum, enable the development of complex applications through the use of Smart Contracts. Despite its increasing popularity, blockchain application development faces many challenges, preventing the widespread adoption of blockchain technologies. The current programming model requires that programmers have a deep understanding of the blockchain’s internal behavior and structures to build correct applications. Additionally, the high commit times lead developers to relax consistency guarantees, often leading to incorrect behaviors and bugs. In this work, we introduce Speculatra, a framework for building efficient correct blockchain-based decentralized applications. Speculatra allows developers to avoid common pitfalls and build more robust applications, while also decreasing latency by speculating on the outcome of transactions.

Our evaluation on the Ethereum network shows that Speculatra can decrease the latency of operations up to 80% when the speculation is successful.

I. INTRODUCTION

While blockchains are popular as cryptocurrency technologies, allowing the transfer of funds without the need for an intermediary, modern blockchain technologies, such as Ethereum [1], provide a way of executing arbitrary programs in the blockchain, known as smart contracts. Smart contracts can be viewed as an implementation of a replicated state-machine (RSM). Each network node maintains a copy of the program state and a state machine that processes transactions. A consensus protocol determines the next state-transition.

Smart Contract applications can be extended to a variety of use cases, including identity management [2] [3], shipment tracking, food safety, dispute resolution, tracing diamond provenance [4], electronic auctions [5] and database systems [6]. DApps are blockchain-based decentralized applications, that allow users to access web pages to interact with smart contracts in the blockchain [7].

Smart contracts [8] consist of immutable geo-distributed computer programs that run in the context of a blockchain platform. They maintain a long-lived distributed state, recorded in state variables, managed through the invocation of functions, either directly by a client submitting a transaction, or indirectly through other contracts. Changes to a smart contract are only visible once the corresponding transaction is included in a mined block. As others have noted [4], smart contracts can support state-machine replication [9] in permissionless Byzantine settings.

To perform an update, client applications first must submit a signed transaction to the blockchain network. If valid, the transaction will eventually be included in a block in the blockchain. However, multiple parallel blocks may be generated at the same time, and only subsequent blocks will determine which one remains part of the blockchain and which one is discarded. As a result, to confidently determine whether or not a transaction has been confirmed, applications need to wait until the block containing the transaction is deep enough in the chain. At this point, it is highly unlikely that the block will be discarded in the future. Hereafter, we say that, at such point in time, the transaction has committed. In Ethereum, the recommended number of confirmations is 12 [10] and for Bitcoin it is 6 [11], which translates approximately to a commit time of 3 minutes and 1 hour, respectively.

Since the last few blocks of the blockchain are susceptible to change, to ensure strong consistency, blockchain applications must always read the state of committed blocks. However, there might be recent transactions issued by the application during its execution, which have yet to be included in the blockchain or are included in one of the most recent blocks. By reading the state from a committed block, the state observed by the application might not yet reflect effects from one or more recent transactions issued by the application during its execution, which would violate the read your writes criteria that is often expected by most application semantics [12]. Therefore, applications must also wait for all their transactions to be committed before reading the blockchain state.

This approach presents many issues. The high commit times result in very high latencies, which are unacceptable for many applications. Additionally, by looking only at the committed state, freshness is also compromised. In Ethereum, this state is about 3 minutes old. This is a considerable window for new updates, which are not reflected in the returned state but which the process has most likely already seen. As a motivating example, let us consider an eCommerce application. A client checks the stock of a particular item and the committed state indicates there are a few units left. As a result, the client issues a transaction, purchasing the product. However, the product has already been out of stock for the last minute, which means the purchase transaction will fail. Furthermore, while this can be achieved under the current programming model, it is far from trivial. Existing implementation and libraries all require that the programmer have a deep knowledge of the blockchain’s internal behavior and data structures. Additionally, it forces developers to mingle application logic with the low-level blockchain internal details, leading to errors and inefficiencies.

In this work, we propose Speculatra, a framework for developing strongly consistent applications in Ethereum that leverages speculative execution to address the high latencies.
and freshness limitations of the current programming model. Speculation is a latency-hiding technique frequently used in distributed systems, which allows applications to speculate on transaction outcome, and hence reduce overall latency by overlapping commit waiting time with further computation. Rather than wait for 12-confirmations, applications may instead speculatively execute further operations using the result of the transaction at 0-confirmations. The speculative period can be used for both off-chain computation, as long as it is reversible, and in-chain computing i.e. issuing transactions and reading the blockchain state. Speculatra also addresses how applications may issue speculative transactions, i.e. transactions that are issued based on a speculative read and, hence, may need to abort if the speculation fails.

Incorrect speculations occur when a blockchain reorganization changes the predicted transaction outcome. Incorrect speculations waste resources since any processing steps that have been executed after the speculative read need to be rolled back and start again. Nevertheless, a preliminary analysis on the real Ethereum blockchain showed that blockchain reorganization, due to blockchain forks, is a relatively rare event [13]. Consequently, the probability that a speculative read is later deemed to be incorrect is also low. Therefore, speculating over smart contract reads in Ethereum is expected to be a very profitable strategy.

Furthermore, Speculatra abstracts the internal details and behaviors of blockchains, leading to simpler and less error-prone code.

Speculatra was implemented on top of Web3.js [14], one of the most popular libraries for interacting with Ethereum smart contracts. Speculatra is compatible with the current implementation of Ethereum and existing smart contracts. Therefore, it can be used by an Ethereum DApp, new or existing.

We evaluated Speculatra in a simulated Ethereum network and in Ropsten [15], an Ethereum test network, which best reproduces the current Ethereum production environment. Our evaluation of Speculatra in Ethereum shows that it can achieve comparable performance to solutions that guarantee weaker consistency, with minimal overhead. We show that Speculatra can provide up to 80% speedups on a representative DApp, with marginal performance overheads in worst-case scenarios where speculations fail. Additionally, Speculatra scales well for an increasing number of transactions, reads, and writers.

II. BACKGROUND

Bitcoin [16] and Ethereum [1] are the two most popular blockchains. We choose to focus on Ethereum because it offers a full general-purpose language, while Bitcoin offers a more restricted scripting language [8].

A. Ethereum as a State Machine

The Ethereum blockchain is a distributed, deterministic state machine [17] [8]. The Ethereum world state consists of a mapping between account addresses and account states and is stored in a modified Merkle Patricia Trie [18]. For simplicity purposes, we will refer to it simply as a trie in the rest of the paper.

Transactions represent Ethereum state transitions [19] and are grouped into blocks.

Ethereum has an intrinsic currency called Ether [19]. An account is an entity associated with a balance that can be used to send transactions. There are two types of accounts in Ethereum: i) Externally Owned Accounts (EOA) are used to send transactions and ii) Contract Accounts [17] are controlled by their contract code [17].

B. Life Cycle of a Transaction

Each transaction included in the blockchain is validated and executed by all nodes in the network. To update the state of the blockchain, a client submits a signed transaction, describing the update, to its local Ethereum node. The node will validate the transaction, before propagating it to all its neighbors and returning the hash of the signed transaction, which can be used to track its status. Each neighboring node, i.e. nodes in the network that share a link, will also validate and propagate the transaction [17]. The Ethereum network is composed of miner and non-mining nodes. Miner nodes maintain a transaction pool to which they add received transactions. The transactions selected to be included in a candidate block, i.e. a block that might be included in the blockchain, are picked from this pool. All transactions from a given originating account must be processed sequentially, according to their nonce.

Once the miner has selected the transactions to include in a candidate block, it attempts to find the Proof of Work (PoW) that makes the block valid [17] [8] [19]. Proof of Work [16] [17] is a consensus algorithm that allows nodes to agree on a set of canonical updates to the blockchain state. Then, the miner will propagate the block to all neighboring nodes, which will validate and propagate it as well [19].

Every transaction included in the block is executed, first by the miner that generated the block and then by every other node in the network. The transaction execution may fail due to an exception or because the transaction aborts. However, regardless of whether the transaction execution succeeds or fails, the stateRoot, which is the hash of the structure that represents the world state, is always modified, the transaction is appended to the block transaction trie and the transaction receipt is appended to the block receipt trie [8]. In Ethereum, miners may choose whichever transactions they wish to include in a block, as long as the gas required by all transactions in a block does not exceed the block gas limit [8]. Gas is the virtual currency charged for the transaction fee given to the miner that successfully mines the transaction.

Once the miner node finds the PoW that makes the block valid, the miner appends it to the block header and broadcasts the block to its neighboring nodes. Nodes that receive the block check that the PoW in the block header and the state transactions are valid before also propagating the block to their neighboring nodes [17]. Miners compete to find the next block in the chain, which means that it is possible for two or
more miner to find the next block in the chain at nearly the same time, causing parallel chains to emerge. This is called a fork [17] [19]. In this situation, blocks with the same height in different forks will probably not contain the same set of transactions, in the same order.

In Ethereum, a fork is eventually resolved for the longest branch. This means that the higher the depth of a block in the blockchain, the smaller it is the probability it might be replaced if a reorganization were to happen, and the greater it is the degree of confidence we have that the result of the transaction execution will not change.

This means that, depending on how we handle concurrent read and write operations, we can support different consistency models. As the number of confirmations increases, the probability of inconsistency decreases exponentially. Once a transaction has 12-confirmations, we can consider it committed [10].

C. Smart Contracts

Smart Contracts [8] [20] are deterministic computer programs that are executed in the context of the Ethereum Virtual Machine (EVM), typically programmed using a high-level, Turing-complete, programming language, such as Solidity [21]. Contracts are compiled into EVM bytecode and then deployed on the blockchain network, using a contract creation transaction that assigns them a unique address [19]. Smart Contracts are immutable, which means their code cannot be modified after deployment.

There are two modes of Smart Contract execution: calls and transactions [22]. Calls are used to evaluate the contract state. Calls are read-only executions of the Smart Contracts. Since a call is only executed by the local node and is not propagated to the remaining Ethereum nodes, it cannot change the blockchain state.

Transactions are used to update the contract state. When a Smart Contract transaction is executed, the EVM runs it within a sandboxed copy of the Ethereum state. Upon successful completion of the contract execution, the world state is updated to match the sandboxed version. If the contract execution fails, for any reason, all changes applied to this sandboxed state are discarded. In either case, a transaction receipt is generated containing the outcome [8].

A contract can invoke another contract through a message call [19], generating an internal transaction. With each message call, another EVM is instantiated with the state from the sandbox of the EVM at the level above [8]. Independently of how many contract invocations a transaction may generate, the outer transaction (which may comprise multiple inner invocations) is atomic. Atomicity [23] guarantees that either all operations contained in a transaction are applied or, if any fails, the changes resulting from the previous operations are discarded and the final state remains the same as the state prior to the transaction execution. Internal transactions are not explicitly recorded in the blockchain.

D. Impact of Transaction Execution Order

The execution order of transactions can impact the final state of the blockchain. If the order of transactions were deterministic, the transaction outcome speculation would be correct with an overwhelming frequency. As it is, a fork might cause the speculation to fail.

Let us consider the following transactions, which interact with the Smart Contract in Listing I: T1:Add(4), T2:Add(2) and T3:Multiply(4). Since additions are commutative, T1 and T2 are commutable. Regardless of the order of execution, the final value of the Smart Contract variable Value will be 6, once T1 and T2 have been executed. In contrast, T1 and T3 are non-commutable, since multiplications and additions are not commutative. Depending on the order of execution, Value will either be 16 or 4.

This means that non-commutable operations that might result in a conflict must wait for the transactions to achieve 12-confirmations and will consequently be slower. In contrast, commutable operations can be executed with more relaxed consistency, (i.e. waiting for less confirmations) and thus faster. Programmers need to be aware of the consistency requirements for each operation to be able to optimize the application performance.

Nevertheless, forks affect only less than 10% of Ethereum blocks [13], hence the speculation will be correct most of the time.

E. Ethereum Stack

Smart Contracts, implemented in a high-level language such as Solidity [21], store the business logic and the application state. Once the contract is deployed, the client-facing DApp interface interacts with the contract to update and read the application state by sending transactions and calling the contract functions using the JSON-RPC [24], provided by the Ethereum blockchain, or a high-level interface, e.g Web3.js [14]. Additionally, it might also use the subscription feature supported by Web3.js by to monitor updates to the blockchain state.

Listing I: Solidity Smart Contract used for evaluation.

```solidity
pragma solidity >=0.4.25 <0.6.0;

contract Calculator {
    int public Value;

    constructor() public {
        Value = 0;
    }

    function Add(int input) public{
        Value += input;
    }

    function Subtract(int input) public{
        Value -= input;
    }

    function Multiply(int input) public{
        Value *= input;
    }

    function Divide(int input) public{
        if(input == 0) revert();
        Value /= input;
    }

    function Clear() public{
        Value = 0;
    }
}
```

1. D. Impact of Transaction Execution Order
2. The execution order of transactions can impact the final state of the blockchain. If the order of transactions were deterministic, the transaction outcome speculation would be correct with an overwhelming frequency. As it is, a fork might cause the speculation to fail.
3. Let us consider the following transactions, which interact with the Smart Contract in Listing I: T1:Add(4), T2:Add(2) and T3:Multiply(4). Since additions are commutative, T1 and T2 are commutable. Regardless of the order of execution, the final value of the Smart Contract variable Value will be 6, once T1 and T2 have been executed. In contrast, T1 and T3 are non-commutable, since multiplications and additions are not commutative. Depending on the order of execution, Value will either be 16 or 4.
4. This means that non-commutable operations that might result in a conflict must wait for the transactions to achieve 12-confirmations and will consequently be slower. In contrast, commutable operations can be executed with more relaxed consistency, (i.e. waiting for less confirmations) and thus faster. Programmers need to be aware of the consistency requirements for each operation to be able to optimize the application performance.
5. Nevertheless, forks affect only less than 10% of Ethereum blocks [13], hence the speculation will be correct most of the time.
6. E. Ethereum Stack
7. Smart Contracts, implemented in a high-level language such as Solidity [21], store the business logic and the application state. Once the contract is deployed, the client-facing DApp interface interacts with the contract to update and read the application state by sending transactions and calling the contract functions using the JSON-RPC [24], provided by the Ethereum blockchain, or a high-level interface, e.g Web3.js [14]. Additionally, it might also use the subscription feature supported by Web3.js by to monitor updates to the blockchain state.

Listing I: Solidity Smart Contract used for evaluation.
Solventy Smart Contracts have access to a set of globally defined variables and functions that can be used to obtain information about the blockchain. The blockhash(uint blockNumber) function, for example, returns the hash of the block at the given depth.

Solventy also allows developers to define custom function modifiers, which can be used to add a pre-condition checks to a contract function. Functions can be modified by including the named modifier in the function declaration.

The two most important methods provided by the JSON RPC API are as follows:

i) `eth_sendTransaction(transactionObject)` allows users to update the blockchain state by submitting transactions.

ii) `eth_call(callObject, blockId)` triggers a local, read-only execution of a contract function. If blockId is omitted, the contract will execute on the most recent local blockchain state, i.e. the state stored in the last block in the local node blockchain. Otherwise, it will execute on the blockchain state stored at the specified block. There are several libraries providing high-level interfaces to these RPC methods.

The `subscribe` method allows subscribing to specific events, such as i) `newBlockHeaders` which sends events for new block and, ii) `pendingTransactions sends` events for incoming pending transactions.

### III. Speculative DApp: Motivation

We now introduce a sequential consistent DApp, and motivate how it can be improved with Speculatra. Let us consider an eCommerce DApp, implemented using the Smart Contract shown in Listing 2.

To add a product P to the cart, a customer must submit a transaction T1 invoking the `addToCart` function (line 25—33). This function will check if there are any available units of the product and reserve a unit for the customer.

To finalize the purchase, we first need to display the cart contents so that the customer can confirm the order. For each product, the DApp displays some product information stored in a remote server (e.g., picture, brand and product description). We start by calling the `getCart` contract function (line 36—47), which returns the current list of products in the cart and their total price. After obtaining the list of products, we fetch the information for each of the products.

If there are transactions issued by the customer updating the contents of the cart that have not yet been committed, the list of products is merely speculative, as the transactions might be reordered. Let us assume that there is only a single unit of product P left in stock, and a different customer has issued T1’, concurrent with T1. At the time the speculative value returned, T1 was ordered before T1’ in the blockchain, which means that the product P is in the customer’s cart. If later a fork results on T1’ being reordered before T1, the customer’s cart will not contain product P. Were the customer to submit a transaction T2 invoking the `checkoutCart` contract function (line 51—53) after seeing the speculative contents of the cart, the customer would ultimately purchase a different set of products than he had believed.

However, if the transaction T2 is a speculative, this is avoided. A speculative transaction can be supported by adding a check before the `checkoutCart` function execution, which guarantees the transaction will fail if the contents of the cart at execution differ from the speculative contents. When a transaction is affected by a fork, its effects are discarded and the transaction execution is repeated. This means that, if a fork affects the speculation, when the transaction is re-executed, it fails the precondition check and aborts. With this approach, no compensatory transaction is required to revert the effects of the speculative execution.

To be practical, this precondition check must be relatively simple and therefore not require much gas. Additionally, this precondition should ideally be generic enough that it can satisfy the majority of use cases. Compared to the first alternative, which required waiting for the transactions to be committed, this one significantly reduces latency, even in cases of mispeculation, because we don’t need to wait for the transaction to be committed. It is also less expensive than the second one presented because issuing T3 is likely more expensive than the added cost of T1 introduced by the precondition check. However, when the speculation is successful, this approach has a higher gas cost compared to the other alternatives. It might not even be feasible if the original contract function already requires close to max gas, which corresponds to the maximum gas for the block in which it is included.

### IV. Speculatra

This section presents Speculatra, a framework for supporting speculation on applications whose semantics are based on sequential consistency. Speculatra reduces latency by allowing applications to speculate on the outcome of transactions that are not yet committed.

The main method offered by Speculatra is:

```
speculate(data, speculateCallback, abortCallback, commitCallback)
```

This method allows developers to speculate on the result of uncommitted transactions. Its arguments are: i) the transaction object for the method to be called; ii) the callback function that will process the speculative result; iii) the callback function that will handle errors and failed speculations and iv) the callback function that handles the speculation confirmation. It returns a `call object`, which we describe in Section IV-A.

Listing 3 shows an example of a DApp using this interface.

The DApp initializes Speculatra (line 2), specifying the address of Smart Contract defined in Listing 2. The function `purchase` (line 4) receives a list of products and tries to purchase all items in the list. To do so it submits, for each product, a transaction T1 invoking the Smart Contract methods `addCart` (lines 5—8). Then, it attempts to read the speculative contents of the cart (lines 9—10). When Speculatra
returns the speculative result, the list of products in the cart is compared with the list of products received as an argument. If the lists contain the same products, a transaction T2 is sent invoking the Smart Contract method `checkoutCart` (lines 11—18). If, later, Speculatra finds out that the speculation failed, the DApp reverts the transaction T2 (lines 19—20). Otherwise, once the speculation is confirmed, the purchase is also confirmed (lines 21—23).

The API introduced before should observe the following set of requirements:

1. Ensure sequential consistency
2. Leverage speculation to reduce the latency of operations
3. Maximize freshness
4. Abstract internal details and behaviors of the Ethereum blockchain

To fulfill these requirements, Speculatra faces many challenges. The first challenge we encounter is how to select the block from which to read the speculative state. To ensure the `read-your-writes` property, we need to read the state from a block B in which the Smart Contract state reflects all the transactions issued during the application’s execution. We also need to consider that some of these transactions might not have been included in a block yet and hence executed. Speculatra must also be able to reliably detect failed speculations. Intuitively this might seem simple, as misspeculations are always a result of a fork, which are fairly easy to detect. Additionally, to increase the probability of successful speculation, we want to read from the most consistent state possible.

To maximize freshness we need to also consider concurrent transactions updating the contract state, which have been issued by other applications. This is not trivial as the current programming model offers no solution for easily filtering transactions that invoke a particular contract. This is especially difficult if the transactions do not invoke the contract directly.

not all forks will cause speculation to fail. The fork might not affect the set of transactions that modify the contract state if they are not included in a block discarded by the fork or appear in the same order in the new blocks. Even if these transactions were affected by the fork, the transaction may be commutable.

Applications can take advantage of the speculative period to execute both in-chain and off-chain operations. However, any updates issued based on the speculation must be reversed if the speculation fails. This is particularly challenging for in-chain operations because Ethereum currently does not provide any mechanism designed for this purpose.

A. Speculatra State

This section provides an overview of the main data structures maintained by Speculatra.

Speculatra maintains a list of the transactions that potentially affect the contract state, which we will refer to as `tracked transactions` moving forward. For each of transaction, Speculatra maintains the following state:

- `hash` — transaction hash
- `state ∈ {PENDING, UNCOMMITTED, COMMITTED}`

Listing 3: Basic eCommerce DApp using Speculatra.
• blockNumber – number of the block containing the transaction
• blockHash – hash of the block containing the transaction
• confirmations – number of confirmations for the transaction

A transaction can be in one of three states: i) pending, which means the transaction has not yet been included in a block in the blockchain, still remaining in the Ethereum node’s mempool; ii) uncommitted, if the transaction is included in a block in the blockchain, but has less than 12 confirmations and finally iii) committed, once the transaction has received 12 confirmations. Once a transaction is committed, it is deleted from the list of tracked transactions.

Each speculation request is represented by a call object, which maintains the following state:
• data – transaction object of the method to be called.
• speculate – callback for processing the speculative result.
• abort – callback for errors and failed speculations.
• commit – callback that handles the speculation confirmation.
• txs – set of transactions it depends on.
• state ∈ \{PENDING, EXECUTED, VERIFY\}
• result – initially None

The set of transactions on which a call depends is defined as the set of tracked transactions which are either in the pending or uncommitted states at the time the speculate method is invoked. For the remainder of this document, we will refer to this set of transactions as relevant transactions.

Speculatra also keeps track of a block history, which contains all the blocks that are included in the blockchain and have not yet been committed.

V. IDENTIFYING SMART CONTRACT UPDATES

To ensure sequential consistency, Speculatra must guarantee that each read returns a value that reflects all updates to the Smart Contract state issued during the application’s execution. Additionally, to increase freshness, Speculatra should also consider updates issued by other applications. Therefore, we need a method for identifying all the transactions which potentially modify the contract’s state.

Transactions can interact with Smart Contracts directly, or indirectly, by executing a contract function that invokes another contract. An indirect invocation of a Smart Contract is not registered in the blockchain. As result, to determine whether a transaction indirectly invokes another contract we need to instrument the EVM. This incurs significant overhead, as every transaction in the blockchain, regardless of whether it actually interacts with the Smart Contract, must be instrumented. Additionally, it requires modifying the EVM, which renders the solution incompatible with current Ethereum clients.

However, if we were able to identify which Smart Contracts invoke the Smart Contract in question, we could avoid instrumenting the transaction’s execution. Instead, we could simply flag the transactions that invoked any of the Smart Contracts that interact with the relevant Smart Contract. This solution incurs significantly less overhead than the previous solution. Nevertheless, it is not without issues.

This imposes on the developer the burden of determining which Smart Contracts interact with the relevant Smart Contract. However, considering the code of all deployed Smart Contracts is publicly visible, and platforms such as Etherscan [25] allow us to see the internal transactions of a contract, we believe it is reasonable to assume that developers can achieve this with minimal effort.

The second issue is that, with this approach, we may be considering a larger set of transactions than those which are truly relevant. This means that the speculation may return later than necessary, as it can only return once all transactions it depends on have been executed. That said, our solution guarantees that, even if these transactions, which do ultimately affect the contract state, are affected by a fork, speculation will not fail as a result, as we will demonstrate in Section VII.

Some applications might be willing to sacrifice freshness to minimize the time it takes for the speculative value be returned. That is, some applications might only be concerned with the results of a specific subset of transactions e.g. the transactions issued by them. With this in mind, Speculatra supports three different variants: i) multiple-writers, ii) selective-writers and iii) application-specific. In the multiple-writers variant, which is the default, Speculatra considers all transactions which interact with the Smart Contract, issued by any client. Transactions are identified with the method we just described.

In the selective-writers mode, Speculatra only considers the transactions that were issued by one of the specified clients. This variant requires that, upon initialization of Speculatra, the application specify the address of the client accounts which should be considered.

Finally, the application-specific variant only considers that transactions the application explicitly registers, using the registerTx, which receives the transaction hash.

VI. SPECULATIVE READ

DApps can speculate on the state of a Smart Contract by calling a contract function. The Smart Contract state is only updated once a transaction has been executed. Therefore, for speculation to reflect the effects of a transaction, the corresponding call can only be executed once the transaction has been included in a mined block and on the state in the block containing the transaction (or one of the subsequent blocks). Additionally, the probability that a block will be discarded due to a future fork decreases the deeper the block is in the blockchain. Therefore, by reading a deeper block, we decrease the probability of a mispeculation.

Algorithm 1 shows the pseudo-code for executing a call and selecting the block on which state the call will be executed. The call is executed only once all transactions it depends on have been mined (line 2). If the call does not depend on any transaction (line 18), then the block at depth 12 is selected (line 19). Otherwise (line 20), the block in which the transaction
Algorithm 1 Execute call

1: Globals: history ← latest blockchain blocks seen

2: Require: for all tx ∈ call.txs : tx.state ≠ PENDING

3: function EXECUTECALL(call)
4: callBlock ← SELECTBLOCK(call)
5: error, result ← await WEB3.CALL(call.data, callBlock)
6: if error then
7: call.abort(error)  ▷ Call failed
8: calls = calls \ call
9: else if result ≠ call.result then
10: call.abort(result)
11: calls = calls \ call
12: else ▷ Speculation did not fail
13: call.state = EXECUTED
14: call.result = result
15: end if
16: end function

17: function SELECTBLOCK(call)
18: if call.txs == ∅ then ▷ Committed block
19: callBlock ← history.latest.blockNumber - CONFIRMATIONS
20: end if
21: callBlock ← MAX(call.txs, lambda key: tx.blockNumber) .blockNumber
22: end function

with fewer confirmations is included is selected (line 21). Executing the call consists of locally executing the transaction described by the transaction object in data on the state stored in the selected block (line 5). Once the call has been executed, its state is set to EXECUTED and the speculative result is stored in result (lines 13-14).

VII. DETECTING MISSPECULATIONS

Speculatra must be able to consistently detect misspeculations. Failed speculations are caused by forks. However, forks do not always cause speculation to fail. If the blocks containing the transactions the call depends on are not discarded, the call is not have been affected. Additionally, if the order of the transactions in the blockchain remains the same or if the transactions are commutable, the call will also not be affected. Hence, to guarantee that the speculation has in fact failed, we need to: i) determine if any of the transactions it depends on have been affected and if so ii) re-execute the corresponding call and compare the result obtained with the previous result.

With this approach, we can detect all failed speculations. However, it might generate false positives. Since we cannot see the intermediate state of a block, even if, after a fork, we have the same set of transactions, in the same relative order, but with a different distribution between blocks, the result of a call after the fork is resolved may differ from the result of the previous execution.

Algorithm 2 shows the pseudo-code for handling a fork. We must first determine which blocks were discarded (line 3) and then identify the tracked transactions included in those blocks (line 4), changing their state to PENDING (lines 5—7). Next, we flag all calls that depend on those transactions by setting their state to VERIFY. Once we verify that all the affected transactions the call depends on have been included in a new block in the blockchain, we execute it again (see Algorithm 1) and compare the result with the previous one (line 9). If the results differ, then we conclude the speculation has failed and trigger the abort callback (lines 9—11).

Algorithm 2 Handle reorganization

1: Globals: uncommitted ← txswith tx.state = UNCOMMITTED
calls ← callsSubmitted
history ← stack latest blockchain blocks seen

2: upon event < reorganization, newBlock > do
3: discardedBlocks ←
UPDATEBLOCKHISTORY(history, newBlock)
4: affectedTxs ←
{tx ∈ uncommitted : tx.block ∈ discardedBlocks}
5: for tx ∈ affectedTxs do
6: call.state ← PENDING
7: end for
8: for call ∈ calls do
9: if call.state = EXECUTED
10: then
11: call.state ← VERIFY
12: end if
13: end for

14: function UPDATEBLOCKHISTORY(history, newBlock)
15: discardedBlocks ← \{}
16: canonicalBlocks ← \{}
17: while history.latest.hash ≠ newBlock.parentHash do
18: discarded ← history.pop()
19: discardedBlocks ← discardedBlocks ∪ discarded
20: newBlock ← getBlock(discarded.number)
21: canonicalBlocks ← canonicalBlocks ∪ newBlock
22: end while
23: history.update(canonicalBlocks)
24: end function

A. Speculative Transactions

Applications can take advantage of the speculative window to execute both off-chain and in-chain operations, such as issuing transactions. However, these must be reversible. The most promising approach, which avoids the need for issuing a compensating transaction when the speculation fails requires that the Smart Contract function invoked by the transaction execute a precondition check, before executing the transaction, to validate if the speculation has not failed. Additionally, this precondition must be generic, that is, independent from the application logic implemented by the Smart Contract, and simple so that it consumes little gas.

As speculation is always caused by a fork affecting block B, from which the speculative state was read, by validating that block B was not discarded we can determine the speculation has not failed. This solution is independent of the application logic, and therefore fits the majority of use cases. Additionally, the check is very simple and hence is not expensive. Finally, it allows applications to execute transactions speculatively which depend on more than one speculation. However, this solution is not without faults. If the block was discarded but the relevant
transactions have not been reordered or are commutable, the speculation does not fail but the transaction does.

Listing 4 contains the definition of a Solidity modifier that implements this precondition. It uses the special function blockHash provided by Solidity (see Section II-E) which returns the blockchain for the block with the given block number.

VIII. Evaluation

In this section, we evaluate Speculatra by answering the following question: i) what is the performance gain that can be achieved using Speculatra, compared to traditional approaches for supporting sequential consistency over Smart Contract state, on Ethereum; ii) what is the computational overhead introduced by Speculatra, compared to solutions that achieve comparable performance results;

A. Testing Environment

We evaluate our proof-of-concept implementation of Speculatra in Ethereum. We implemented an Ethereum simulator, using the reference Geth 1.7 implementation [26]. To run experiments with a large number of nodes, we replaced the PoW component with a probabilistic mining selection process that follows a Poisson distribution and mimics the block production distribution. We adjusted the block production probability (by setting the Poisson event rate) to have approximately 3 blocks committed per minute (i.e., a 20-second block time) [25].

For experimental purposes, we also developed a custom client that, using the regular transaction creation API, injects transactions in the system following a real trace obtained from Etherscan [25] (from block 0x50f8f4 up to block 0x5ef20).

We ran each of the experiments described in sections VIII-C and VIII-D using 100 nodes co-located in 5 machines equipped with a mix of Intel(R) Xeon(R) CPUs E5506 @ 2.13GHz - 8 Cores. Each node started with the same blockchain, consisting of a single genesis block. We injected 8 transactions per second, as common in Ethereum [25].

We used the Linux top command [27] to obtain information about the CPU and memory utilization during the experiments.

B. Workload

To evaluate Speculatra, we implemented a DApp that interacts with the Smart Contract defined in Listing 1, which supports the basic functionality of an calculator. The Smart Contract state is reflected in a single state variable Value (line 4), which can be read using the automatically generated getter Value.

In every experiment, we assume an execution where the DApp starts by issuing a transaction T1, invoking Add(2), followed by another transaction T2, invoking Multiply(2). The transactions are submitted from different accounts, to simulate concurrent transactions. Immediately after issuing the transactions, the DApp reads the value of the Smart Contract. The DApp processes the speculative value using a function which returns a boolean. If the processing result is True, the DApp issues a transaction T3, invoking Divide(2). Once T3 is confirmed, the execution is complete.

We decompose this execution in two phases. The first phase terminates once the speculative value is returned. The second phase includes the remainder of the execution.

We dedicate Section VIII-C to evaluate the first phase of the DApp’s execution. Then, in Section VIII-D we evaluate the overall latency of the application’s execution.

C. Speculative Reads

In this section, we focus on the evaluation of the first phase of the DApp’s execution, and evaluate the latency and overhead of read operations using Speculatra.

1. Read Latency

To evaluate the read latency of Speculatra, we considered four different read policies:

(A) Read-immediately. Reads as soon as transactions have been successfully propagated in the network.

(B) Read-after-mined. Reads once both transactions have been mined.

(C) Read-after-committed. Waits for both transactions to be included in a block before reading.

(D) Speculatra. Uses Speculatra, submitting the speculation request once both transactions hashes have been returned. Only the last two policies guarantee sequential consistency. Additionally, we considered the following scenarios:

(I) Same-order. Both transactions are included in blocks not affected by reorganizations, hence, the speculation is successful.

(II) At least one of the transactions is included in a block affected by a reorganization.

(a) Reorder-safe. The reorganization does not affect the final state, in the case of Speculatra, the speculation is successful.

(b) Reorder-conflict. The reorganization affects the final state, therefore, in the case of Speculatra, the speculation fails.

All results are the average of 60 runs, each approximately 20 minutes long.

Figure 1 shows the average read time (R) for each of the policies across the different scenarios analyzed. We define read time as the time between successfully issuing both transactions and the final read result is returned according to each policy.

Listing 4: Solidity Smart Contract that defined a modifier which implements the precondition check that verifies the speculation.

```solidity
contract speculatra{
    modifier verifySpeculation(uint _bNr, bytes32 _bHash) {
        require(blockhash(_bNr) == _bHash,
            "speculation doesn’t exist.");
    }
}
```
If the speculation is correct, we consider the time at which the speculative result is returned as the read time of policy D. If speculation fails, we consider the time at which the speculation failure was detected and the new speculation result was returned.

In the scenarios analyzed, a reorganization occurs at most once from the point at which the transactions are broadcast until they are committed. However, in real-world scenarios, it is possible for more reorganizations to occur, affecting once again the speculation.

As expected, the more relaxed policies read-immediately and read-after-mined present the lowest read times. When comparing policies C and D, which ensures sequential consistency, policy speculatra presents on average lower read times across all the scenarios analyzed. In fact, when the speculation is correct, policy speculatra presents only slightly higher read times than policy read-after-mined (around 1.30 seconds). When the speculation fails, policy speculatra still performs better on average than policy read-after-committed. This is due to the fact the reorganization is typically detected before the transaction is committed in either parallel chain. While this is a synthetic scenario, we can foresee even better results in the real Ethereum blockchain since 97% of reorganizations affect only 1 block [13], which is smaller than the depth of the reorganization in our simulations (~10 blocks).

The policies read-after-committed and textitspeculatra show the lowest CPU utilization. This is because they are blocked waiting for the transactions commit and, consequently, the CPU is not utilized for a considerable portion of their execution.

The more relaxed policies consumed less memory than the other two policies. This was expected since policies read-immediately and read-after-mined must maintain less memory than the sequentially consistent policies read-after-committed and speculatra.

2. Scalability

To evaluate the scalability of our solution, we considered a generalization of the first phase of the execution of the DApp described in Section VIII-B. In every experiment, we assume an execution where the DApp issues \( n_1 \) transactions per second, invoking one of the randomly selected Smart Contract methods. The transactions are submitted from \( n_3 \) different accounts, each account simulating a different writer.

Additionally, the DApp uses Speculatra to read the Smart Contract value at \( n_2 \) times per second.

For this evaluation we considered three different vectors:

1) Number of writers \( (n_1 = 1, n_2 = 1, n_3 \in \{1, 2, 4, 8\}) \)
2) Frequency of transactions \( (n_1 \in \{0.5, ..., 4\}, n_2 = 1, n_3 = 1) \)
3) Frequency of reads \( (n_1 = 1, n_2 \in \{0.5, ..., 4\}, n_3 = 1) \)

In the experiments with a single writer \( (n_3 = 1) \), the DApp used the application-specific variant of Speculatra, and the multiple-writers variant was used in the experiments with multiple writers \( (n_3 > 1) \).

For this evaluation, we used the configuration described in Section VIII-A, with no additional modifications to the Ethereum nodes. For each configuration, we collected approximately 250 data points, from a minimum of 10 runs, each taking 20 minutes. For some configurations, we did more runs because the initial results were inconclusive. The results of this evaluation can be seen in Figure 2.

As expected, read times and resource consumption-increase with the number of transactions per second. This is because monitoring the state of transactions is the most computation and memory-intensive task performed. However, the number of reads per second and the number of writers do not have a significant impact on the performance of our solution. The number of reads also does not have a significant impact because the state maintained for a call is minimal. The task in which the number of reads could have a more substantial impact would be handling a reorganization, but reorganizations are infrequent.

3. Speculative Transactions Overhead

We analyzed the overhead of speculative transactions supported by Speculatra, for new and existing Smart Contracts. For existing Smart Contracts, speculative transactions consume 17% more gas for existing Smart Contracts and for new contracts, only 6% more. This difference is justified by the fact that calling other Smart Contract is expensive, as it required loading the second Smart Contract to memory. The Smart Contract functions analyzed were relatively simple and therefore inexpensive. The impact of speculative transactions is smaller if the original smart functions are more complex and expensive.
D. Overall Performance

In this section, we focus on the evaluation of the overall execution of the DApp, considering both phases, and determine the overall performance gain Speculatra can achieve.

For each experiment, we considered four different variants of the DApp described in Section VIII-B:

- **Relaxed.** Follows read policy read-after-mined. This represents the more relaxed approach, which does not guarantee sequential consistency. This is the baseline for Speculatra.
- **No Speculation.** Follows read policy read-after-committed. This represents the traditional approach for ensuring sequential consistency, relaxed approach.
- **Speculatra Light.** Follows read policy speculatra. However, it does not take advantage of speculative transactions. Instead, it only issues T1 once the speculation has been confirmed.
- **Speculatra.** Follows read policy speculatra. Takes advantage of speculative transactions. This approach represents the full Speculatra solution.

We did not consider an approach based on the read policy read-immediately because its behavior would be incorrect almost always and therefore it does not make sense from an application’s perspective.

Additionally, we assume that each scenario described in Section VIII-C occurs with a probability \( P(\text{same-order}) \), \( P(\text{reorder-safe}) \) and \( P(\text{reorder-conflict}) \), respectively. Considering the read and commit times obtained in the evaluation from Section VIII-C, we plotted the total execution time in terms of the processing time.

Figure 3 show the results for \( P(\text{I}) = 90\% \), \( P(\text{II.a}) = 5\% \) and \( P(\text{IIb}) = 5\% \).

IX. Conclusion

We presented Speculatra, a framework for developing blockchain applications with strong consistency semantics. Speculatra leverages speculation to reduce the impact of blockchains high commit times, allowing applications to achieve comparable performance to that achieved by applications with much weaker consistency guarantees. In this work, we showed how new and existing Smart Contracts can be modified to support speculative transactions. Speculatra effectively isolates the low-level details of blockchain state management from the application logic, resulting in more efficient and less error-prone code.

Our evaluation of Speculatra in Ethereum showed that it can reduce the read latency by more than 85\%, when compared to the traditional approach for support sequential consistency, if the speculation is correct.

Overall, we believe that Speculatra is an attractive solution to a long-standing problem in blockchains by allowing applications to achieve lower latencies a a low cost, while ensuring strong consistency.
REFERENCES


