Correct Smart Contract Speculation By Design

Leonor Filipa Dias Alves Parreira Loureiro

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. João Pedro Faria Mendonça Barreto
Prof. Miguel Ângelo Marques de Matos

Examination Committee
Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. João Pedro Faria Mendonça Barreto
Member of the Committee: Prof. Alysson Neves Bessani

January 2021
Thinking doesn’t guarantee that we won’t make mistakes.

But not thinking guarantees that we will.

- Leslie Lamport
Acknowledgments

I would like to start by expressing my deepest gratitude and appreciation to my advisors, João Barreto and Miguel Matos, for accepting me as their student and for their support and guidance during this dissertation. The countless meetings and discussions we had were crucial for the elaboration of this thesis.

Secondly, I would also like to thank Paulo Silva, for all the support and for providing very useful feedback, which was essential during the elaboration of this work. Additionally, I was able to use resources from INESC, which hugely benefited this work and for which I am grateful.

During these past five years at IST, I met incredible people and I am lucky to call some of them friends. A special thank you to André Fonseca, António Terra and João Santos, for their friendship and academic support over the years. I was also part of TFIST - Tuna Feminina do Instituto Superior Técnico, where I had the opportunity of working with wonderful people on projects of which I am very proud.

Finally, I would like to thank my family, especially my mother, Carla Loureiro, my grandmother, Genoveva Parreira, and my boyfriend, Manuel Xarepe, for all their support, care and encouragement over the years.
Resumo

Com o aumento da popularidade das blockchains, as suas aplicações têm vindo a expandir para além do do contexto de criptomoedas. Blockchains modernas, como o Ethereum, permitem o desenvolvimento de aplicações complexas através de smart contracts. Apesar da sua crescente popularidade, o desenvolvimento de aplicações sobre blockchains enfrenta muitos desafios, impedindo a adoção generalizada de blockchains. O modelo de programação atual requer que os programadores possuam profundo conhecimento sobre os protocolos e as estruturas internas da blockchain, para construir aplicações corretas. Além disso, os elevados tempos de confirmação fazem com que os programadores relaxem as garantias de consistência, típicamente levando a comportamentos incorretos. Neste trabalho, apresentamos a Speculatra, uma framework para a construção de aplicações descentralizadas baseadas em blockchain, corretas e eficientes. A Speculatra permite que os programadores evitem os percauçãos comuns e desenvolvam aplicações mais robustas, ao mesmo tempo que contribui para reduzir a latência da aplicação especulando sobre o resultado das transações que ainda não foram confirmadas.

A nossa avaliação do Speculatra, em Ethereum, demonstra que Speculatra consegue diminuir a latência das operações até 80%, quando a especulação está correta.

Palavras-chave: Blockchains, Smart Contracts, Consistência forte, Execução especulativa, DApps
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.

Keywords: Blockchains, Smart contracts, Strong consistency, Speculative execution, DApps
# Contents

Acknowledgments ................................................................. v
Resumo ................................................................. vii
Abstract ................................................................. ix
List of Tables ............................................................... xiii
List of Figures ............................................................... xv
List of Algorithms ........................................................... xvii
Listings ................................................................. xix

1 Introduction ................................................................. 1
  1.1 Thesis Outline ................................................... 4

2 Background and Related Work ........................................... 5
  2.1 Replication and Consistency Models ................................ 5
    2.1.1 Strong Consistency ....................................... 5
    2.1.2 Weak Consistency ....................................... 6
    2.1.3 Hybrid Consistency ....................................... 7
    2.1.4 Implementations ....................................... 9
    2.1.5 Transactional Consistency Models ......................... 9
  2.2 Blockchains and Smart Contracts .................................. 10
    2.2.1 Ethereum as a State Machine ............................. 11
    2.2.2 Ethereum Accounts ..................................... 11
    2.2.3 Transactions as State Transitions ....................... 12
    2.2.4 Impact of Transaction Execution Order ................... 18
    2.2.5 Ethereum Stack ...................................... 19
  2.3 Related work ..................................................... 25
  2.4 Speculative Execution ............................................. 27
  2.5 Discussion ....................................................... 28
List of Tables

3.1 Comparison of three different approaches for issuing transactions based on the Smart Contract state. ......................................................... 32

4.1 Comparison of three variants supported by Speculatra, which differ in how identify tracked transactions. ...................................................... 43

5.1 Comparison of gas cost of traditional transactions with the gas cost of speculative transactions .............................................................. 64
List of Figures

2.1 Overview of the presented consistency models. ........................... 10
2.2 Ethereum’s block and trie structures. ...................................... 12
2.3 Lifecycle of a transaction with a contract account as a destination address. 14
2.4 Diagram that represents a fork in the blockchain with two parallel chains with different final states. ................................ 18
2.5 Traditional DApps development stack. ...................................... 20
2.6 Contents of a transaction receipt. ............................................ 23

4.1 Speculatra DApps development stack. ...................................... 36
4.2 Speculatra transaction state diagram. ...................................... 39
4.3 Speculatra calls state diagram. .............................................. 40
4.4 Diagram illustrating how Speculatra selects the block from which to read the speculative state. ...................................... 44
4.5 Diagram illustrating a scenario in which Speculatra falsely identifies a mispeculation. 45

5.1 Flowchart of the execution of the DApp considered for the evaluation. .... 55
5.2 Illustration of the scenarios considered for the evaluation .................... 57
5.3 Read time and resource consumption of the four DApps compared. ........ 58
5.4 Read time and resource consumption response to an increase in the number of simultaneous writers. .................................. 60
5.5 Read time and resource consumption response to an increase in the number of transactions per second. .............................. 60
5.6 Read time and resource consumption response to an increase in the number of reads per second. ........................................ 61
5.7 Read time and resource consumption with varying pooling timeouts. ....... 61
5.8 Read time and resource consumption of the three different timeout policies compared. .................................................. 62
5.9 Execution Time of a Speculatra DApp, with a 5% probability of mispeculations. 64
5.10 Execution Time of a Speculatra DApp, with a 2% probability of misspeculations. 65
5.11 Execution Time of a Speculatra DApp, with a 8% probability of misspeculations. 66
List of Algorithms

1  Execute call ................................................................. 43
2  Handle reorganization ................................................... 46
Listings

2.1 Snippet of a JavaScript application sending a transaction invoking a contract function. *Adapted from: [52].* ......................................................... 24

2.2 Snippet of a JavaScript application reading data from a contract. *Source: [52].* . 25

3.1 Solidity Smart Contract which implements some basic functionality of an eCom-
merce DApp. ............................................................................. 33

4.1 Basic eCommerce DApp using Speculatra. ............................................. 37

4.2 Solidity Smart Contract that defined a modifier which implements the precondi-
tion check that verifies the speculation. ................................................ 48

4.3 Speculatra *variant* Smart Contract for Solidity Smart Contract in Listing 3.1 . . 50

4.4 Solidity Smart Contract wrapper for the Smart Contract in Listing 3.1 which supports speculative transactions. ............................................. 51

4.5 Basic eCommerce DApp using Speculatra to interact with Smart Contract in 
Listing 4.4. ................................................................................. 51

5.1 Solidity Smart Contract used for evaluation. ............................................. 54

5.2 Speculatra *wrapper* Smart Contract for Solidity Smart Contract in Listing 5.1 . . 67

5.3 Speculatra *variant* Smart Contract for Solidity Smart Contract in Listing 5.1 . . 68
Chapter 1

Introduction

Since the introduction of Bitcoin, the first decentralized cryptocurrency developed by Satoshi Nakamoto, technologies based on the same fundamental principles have become increasingly popular. These technologies are referred to as blockchains [1]. Blockchains are decentralized, peer-to-peer networks, where each participant records transactions in a tamper-proof distributed ledger, stored as a collection of blocks that are cryptographically linked together. Consensus mechanisms ensure that all nodes in the network agree on the same blocks, even in the presence of failures or malicious nodes [2][1].

While blockchains are popular as cryptocurrency technologies, allowing the transfer of funds without the need for an intermediary, modern blockchains, such as Ethereum [3], 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 node maintains a copy of the program state and a state machine that processes transactions. A consensus protocol determines the next state-transition.

Smart contracts [2] 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 [5] in permissionless Byzantine settings.

Smart Contract applications can be extended to a variety of use cases, including identity management [6][7], shipment tracking, food safety, dispute resolution, tracing diamond provenance [4], electronic auctions [8] and database systems [9]. DApps are blockchain-based decentralized applications, that allow users to access web pages to interact with Smart Contracts in the blockchain [10].
To perform an update, client applications first must submit a signed transaction. If valid, the transaction will eventually be included in a block in the blockchain. However, multiple parallel blocks may be generated for the same height, 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, and hence, it is highly unlikely that the block will be discarded in the future. Hereafter, we say that, at such point in time, we say that the transaction has committed. In Ethereum, the recommended number of confirmations is 12 [11] and in Bitcoin it is 6 [12], which translates approximately to a commit time of 3 minutes and 1 hour, respectively.\(^1\)

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 more 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 [13]. 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 client 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 that 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 [14][15] 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.

To mitigate the performance limitations of strong consistency, some blockchain applications

\(^1\)We note that different terms can be found in blockchain literature, such as confirmed, persistent or stable.
opt for weaker consistency models, not waiting for transactions to be committed before evaluating its result and/or confirm it. These weaker models provide significantly better performance but render the programming model more complicated. Let us consider, as an example, a social media application. Minds [16] is a social media platform based on blockchain which provides all the main features of popular social media applications such as Facebook, Twitter, and YouTube. Operations in social media applications must always complete with low latencies as to support the *always-on* experience the users have come to expect. However, weak consistency semantics could result in a confusing user experience if, for example, a user replies to a comment, and at a later point in time, the reply is visible but the original comment is not.

In brief, developers struggle to balance consistency and performance of DApps due to the high *commit times* and the complex programming model supported by blockchains. 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 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 restart. Nevertheless, a preliminary analysis on the real Ethereum blockchain showed that blockchain reorganization, due to blockchain forks, is a relatively rare event [17]. Consequently, the probability that a speculative read is later deemed to be incorrect is also low. Therefore, speculatively 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* [15], 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 new and
existing Ethereum DApps.

We evaluated Speculatra in a simulated Ethereum network and in Ropsten [18], 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.

1.1 Thesis Outline

The remainder of the document is structured as follows. Chapter 2 explains how blockchain technologies work and how Smart Contracts are executed and presents different consistency models that have appeared in the context of distributed systems over the years. It also discusses how speculative execution is used in traditional distributed systems to reduce latency and presents a body of related works that aim to better support Smart Contract development to improve the performance of Smart Contracts execution. Chapter 3 motivates Speculatra and discusses a real-world application of Speculatra. Chapter 4 discusses the design choices we made and presents the implementation of Speculatra. Chapter 5 describes how we evaluated our solution and discusses the results our evaluations. Finally, Chapter 6 concludes the document.
Chapter 2

Background and Related Work

In this chapter, we will discuss existing work in the area of blockchains and Smart Contracts, consistency models designed for traditional distributed systems and also applications of speculative execution to optimize the performance of traditional distributed systems.

We will begin by discussing different consistency models, with emphasis on non-transactional consistency models. Then we provide some background on the basics of the blockchain internal behavior and structures, discuss the life cycle of a transaction and the details of Smart Contract executions and finally discuss the programming model currently offered by Ethereum. Finally, we discuss existing work in the area of Smart Contract development and execution, as well as in the development of decentralized blockchain applications (DApps). We conclude this section with a discussion of the consistency model offered by blockchain, the potential of speculative execution to reduce latency and address the limitations of Ethereum’s programming model.

2.1 Replication and Consistency Models

The notion of consistency is usually defined in the context of replicated systems in which a group of logical objects is replicated in multiple processes, typically deployed at different locations, to improve performance and availability and ensure fault-tolerance. Next, we discuss some consistency models present in the literature [19].

2.1.1 Strong Consistency

Ideally, a distributed system should be a scalable, fault-tolerant version of a centralized system that provides the illusion of sequential execution. Such strong consistency requirements are provided by linearizability. Linearizability [20] implies that each operation shall appear as if
it was applied instantaneously in every replica at a certain point in time, in an order consistent with the real-time ordering of the operations.

The strong semantics required by linearizability make it very challenging to efficiently implement it. **Sequential consistency** [21] relaxes the properties offered by linearizability as it does not require real-time ordering of operations across sessions, only that the ordering of operations for each process is preserved.

From a programming perspective, strong consistency models are the most simple as they provide the illusion of sequential execution of operations observed in centralized systems.

### 2.1.2 Weak Consistency

The goals of high availability and strong consistency have been proved as conflicting in many practical circumstances. The **CAP theorem** states that, in the presence of a network partition, it is impossible for a distributed system to simultaneously guarantee consistency and availability. The proliferation of web services has led system designers to embrace availability and partition tolerance at the cost of strong consistency by leveraging distribution and replication to improve the scalability of these systems and reduce latency for client operations.

At the other end of the consistency spectrum, we have **weak consistency**, which, in practice, does not provide any ordering guarantees, making it so that it has limited usability in real-life systems. Even so, weak consistency can still be useful in contexts where synchronization protocols prove to be too expensive, and an occasional exchange of information between replicas is enough. Systems with high availability requirements often choose to adopt a slightly stronger model, **eventual consistency** [13]. Eventual consistency guarantees that, if a data item receives no more updates, eventually all replicas will converge towards the same state.

The lack of guarantees regarding the order of operations in weakly consistent systems can lead to a confusing user experience. For example, in a messaging service, a user reads a message and responds to it and later he can see the response message but not the message he was responding to. To alleviate this problem, it is useful to define consistency guarantees in the context of a client session. A session is an abstraction for the set of read and write operations performed during an application’s execution. **Session guarantees** [13] are useful to provide a client with a view that is consistent with his actions. There are four types of session guarantees:

i) **monotonic reads**, which states that once a client has read a value \( v \), all successive read operations cannot return a value written before \( v \); ii) **read-your-writes**, which states that a client can only read from a replica which has applied all writes the client has performed so far; iii) **monotonic writes**, which ensures that writes are only handled by replicas that have applied
all writes from the same session; and finally iv) **writes-follow-reads**, that requires that writes 
invoked during a session are ordered after any writes invoked by other clients, whose effects were 
seen during the session.

**FIFO (PRAM) consistency** [22] does not provide a global-ordering of operations but 
guarantees that all updates issued by a given process are seen by all processes in the order in 
which they were invoked, and **per-key sequential consistency**, which ensures a global order 
for all operations on the same key.

**Causal consistency** [23] guarantees that two operations $a$ and $b$ are ordered if they are 
both part of the same thread of execution, also if ii) $b$ reads a value written by $a$, or if iii) they 
are related by transitive closure. Consequently, causal consistency implies FIFO consistency. 
This model allows two concurrent updates to be replicated in any order. If, for example, each 
update maps a different value to the same key in a key-value store, then they are in conflict. 
**Causal+ consistency** [24] strengthens this model by requiring that all replicas handle updates 
in the same way, using an associative and commutative handler function. The *last-writer-wins* 
rule is a common way to resolve conflicting updates. According to this rule, the update that is 
considered more recent overwrites the other.

### 2.1.3 Hybrid Consistency

Stronger consistency semantics are much easier to understand however, they came at a cost to 
performance. A common argument in favor of weaker consistency models is that inconsistencies 
are rare and, in most cases, weakly consistent values are accurate. However, some systems 
cannot risk inconsistencies, forcing them to resort to stronger models. Other systems, value 
performance the most and are willing to risk observing anomalous behavior even if sparingly. A 
third class of systems has conflicting requirements of performance versus consistency. In some 
cases, they must guarantee strong consistency, while in others they can sacrifice consistency 
to achieve better throughput. **Hybrid consistency** models provide the flexibility required by 
these systems.

**Eventual serializability** [25] classifies operations as strict or non-strict. Strict operations 
require all preceding operations to be totally ordered before returning while non-strict operations 
may be reordered after the initial response is returned. Requests are accompanied by a list of 
identifiers of operations that must be ordered before the requested operation and a flag that 
indicates the type of operation.

**RedBlue** [26] consistency categorizes operations as blue operations, which are fast and 
eventually consistent, and red operations, that are strongly consistent but slow. All blue op-
erations must commute with all other operations, blue or red. To increase the number of blue operations, we can decompose most non-commutable operations into commutable operations. RedBlue splits each original operation into a generator operation that is only executed at the primary site, has no side effects and produces a shadow operation. The generator operation determines which state transitions the original operation produces and the shadow operation defines how to apply those transitions in a state-independent manner so that, if applied to the same state as the original operation, both result in the same final state. However, since the state transitions are chosen accordingly to the local state at the primary site and not according to the system state at the time they are applied, the final state is not guaranteed to match the final state for a serial ordering of the original operations.

**Incremental consistency** [27] takes advantage of the fact that, in most cases, preliminary, speculative values are correct by leveraging the use of correctables. Correctables are based on promises, but instead of representing a single value, they capture successive refinements on the result of an ongoing operation. While waiting on the final, consistent result, applications can perform further processing based on the speculative value that was first obtained. If the preliminary result was correct, the application effectively reduced the latency required by strong consistency.

Initially designed for ad-hoc gaming, **vector-field consistency** [28] demands that, for each object, a three-dimensional consistency vector be defined, describing the maximum divergence allowed between replicas in terms of time, number of updates and the object’s value. Vector-field consistency aims to find a balance between bandwidth usage and playability by taking advantage of the fact that positions closer to the player in the game world have stronger requirements in terms of freshness, frequency of updates and accuracy than those that are farther away.

**Consistency rationing** [29] differs from traditional consistency models by allowing developers to define consistency requirements on the data instead of on the operations. Data is rationed into three different categories (A, B or C) according to the business costs of transactions and inconsistencies. Category A offers the strongest consistency level of the three, serializability. Consistency violations for data in this category came at a high cost. The weakest consistency model is offered to data in category C and only guarantees read-your-writes monotonicity for the duration of a client session. Category B contains all data with adaptive consistency requirements and switches between session consistency and serializability at runtime. Policies determine when a switch is necessary. Following the general policy, the switch occurs when the probability of inconsistency is high enough that its potential cost supersedes the cost of consistency. The time policy switches the consistency level when a certain, predefined, point in time is reached. Other
policies are restricted to numeric data types, such as the fixed threshold policy, which evaluates the switch based on the absolute value of the data item.

2.1.4 Implementations

Figure 2.1 presents an overview of the consistency models we have discussed so far. Below are examples of systems that implement some of these models.

The **Bayou system** [30] is an example of a highly available system that was designed to support eventual consistency. In Bayou, updates are applied and recorded immediately at whichever replica they reach and marked as tentative. Tentative updates are eventually ordered according to a canonical ordered and marked as committed, but until then they may be undone and reapplied in a different order. Replicas managers exchange updates in pairs, and once updates of two replicas are merged, the replicas managers detect and resolve conflicts using domain-specific logic.

The **Gossip architecture** [31] is an example of a highly available system that supports causal consistency. It provides two types of operations: queries, which are read-only operations and updates, which modify but do not read the state. Nodes periodically exchange *gossip messages*, in the background, to inform each other of the update they have received. Clients can send both operations to any available replica that can provide reasonable response times, but they indicate the set of previous operations on which the current operation is dependent to guarantee that the nodes only return values that reflects at least the updates that the client has seen so far and that they wait to apply the updates until all previous ones have been applied.

**COPS** [24] is a system that provides causal+ consistency, designed to support complex online applications hosted across a small number of large-scale data-centers. COPS considers a key-value data store with two basic operations: get and put, respectively equivalent to read and write operations in a shared-memory system. Different values of a key are characterized by version numbers and COPS replicas always return non-decreasing versions of a key in response to get operations. To guarantee that all replicas independently handle conflicting put operations, as they receive them, in the same way, replicas use an associate and commutative handler function. The default handler function in COPS uses the last-writer-wins rule.

2.1.5 Transactional Consistency Models

The models described above were designed for **non-transactional distributed storage systems**, which deal with individual read and write operations. In **transactional distributed systems** [32], read and write operations are logically grouped in atomic transactions, which
Figure 2.1: Overview of the presented consistency models. *Adapted from: [19].*

means that, if any of the transaction’s operations violate the system’s consistency rules, the entire transaction must fail. Some well-know strong transactional consistency models are, in decreasing order of consistency, strict serializability, serializability and snapshot isolation. Weak models include, also by decreasing order of consistency, causal consistency, strong eventual consistency, and eventual consistency.

**Strict serializability** (SSER) [33] ensures that the concurrent execution of transactions is equal to some sequential execution of the transactions consistent with the real-time ordering of operations. **Serializability** [33] is slightly more flexible as it imposes some total order but removes the real-time, or even per-process ordering, constraints. Unlike the two previous models, **snapshot isolation** [34] does not enforce a total order, allowing sub-operations of a transaction to interleave with the sub-operations of another transaction, however, a transaction executed under snapshot isolation may only successfully commit if the values that it updated have not been modified by any other transaction since the transaction started.

Even though blockchains contain transactions, their execution is always sequential, so we do not have to worry about the interleaving of sub-operations of different transactions. This means that we can consider the application of non-transactional consistency models to the blockchain.

### 2.2 Blockchains and Smart Contracts

A blockchain is a decentralized, peer-to-peer network where each participant maintains a shared-ledger of transactions, stored as a collection of blocks that are cryptographically linked together. They rely on consensus mechanisms to ensure that all nodes in the network agree on the same
blocks [2][1].

Bitcoin [35] and Ethereum [3] are the two most popular blockchain technologies to date. However, they serve two very different purposes. Bitcoin offers a more restricted scripting language which is limited to the evaluation of simple true or false conditions. Ethereum is mainly a platform for developing decentralized, distributed applications. For these reasons, we will be focusing solely on Ethereum [2].

2.2.1 Ethereum as a State Machine

The Ethereum blockchain is basically a distributed, deterministic state machine [36][2]. The Ethereum world state consists of a mapping between account addresses and account states (Section 2.2.2) and is stored in a modified Merkle Patricia Trie [37]. For simplicity purposes, we will refer to it simply as a trie in the rest of the document.

In a Merkle Tree [36], the leaf nodes contain the hash of a block of data, and the non-leaf nodes contain the cryptographic hash of its two children nodes. Since each parent node's hash depends on the hashes of its children, any change to a data block will reflect on a change to each parent node, meaning that the root node of the Merkle Tree is cryptographically dependent on all the data stored in the tree. This property makes it very easy to verify the data integrity of the tree as we need only compare the root nodes of two trees in order to determine if they represent the same state.

2.2.2 Ethereum Accounts

There are two types of accounts in Ethereum: Externally Owned Accounts (EOA) are associated with a private key and have no code associated with them; Contract Accounts do not have a private key associated with them and are controlled by their contract code [36]. All account’s state is stored in the world state trie, which is represented in Figure 2.2.

An account state is composed of [38]:

- an **ether balance**, which represents the number of Wei (1 Ether = $10^{18}$ Wei) owned by the account. Wei is the smallest denomination of Ether;

- a **nonce**, which represents the number of transactions successfully sent from the accounts, in case of an EOA, or the number of contracts successfully created by the account, in case of a contract account;
• the **storage root**, which encodes the hash of the storage contents of the account, and it is only used by contract accounts;

• the **EVM’s code hash** of the account, which is only used by contract accounts.

### 2.2.3 Transactions as State Transitions

Transactions represent Ethereum state transitions [38]. A transaction is a serialized binary message that contains a nonce; the gas price that the originating account is willing to pay; the gas limit, which is the amount of gas the originating account is willing to pay for the transaction; the destination address of the transaction; the amount of ether to be transferred; the components of the Elliptical Curve Digital Signature Algorithm (ECDSA); and the data payload which is a hexadecimal serialized encoding of a function selector, used to identify the contract function invoked, and the function arguments [38].

Because transactions must be signed, they can only be initiated by externally owned accounts since contract accounts are not associated with a private key. Contract accounts, however, can respond to transactions initiated by EOAs by calling other contracts [2].

We can identify three types of transactions in Ethereum [38][2]:

1. A funds transfer between two EOAs;

2. A contract deployment, which creates a contract account;
3. A message call which invokes a contract function, resulting in a change of the contract’s internal state.

Transactions are grouped into blocks. Every block has a header that contains the hash of its predecessor. This makes it difficult to tamper with a block, as it requires that the attacker also modifies subsequent blocks, which can only be successfully accomplished if the attacker controls 51% of the network’s computational power [40].

The block header also stores three different trie structures [38], as shown in Figure 2.2:

- the **stateRoot**, which is the hash of the root node of the trie that represents the world state, after all transactions have been executed and the state modifications have been applied;

- the **transactionsRoot**, which is the hash of the root node of the trie that records the request vectors of all the transactions included in the block (i.e. the account nonce, the gas price and limit, the destination account address, etc.);

- the **receiptsRoot**, which is the hash of the root node of the trie that contains the receipts of all the transactions in the block, which record the outcome of the transaction (i.e. the post-transactional state, cumulative gas used, etc.).

The path to a transaction in the transaction trie and its corresponding receipt in the receipt trie is determined by the index of the transaction in the block. Once mined, the position of the transactions in the block is never modified, and therefore, neither are the block’s transaction and receipt tries.

The life cycle of a transaction, from the moment it is submitted to a local node and until the network nodes validate the new block containing the transaction is described in Figure 2.3:

To update the state of the blockchain, a client submits a signed transaction, describing the update, to his local Ethereum node (step 1 in Figure 2.3). The node will then validate the signed transaction, to ensure it was signed by the sender’s account, before transmitting the transaction to all other nodes directly connected to the originating node (steps 2 and 3 in Figure 2.3) and returning the hash of the signed transaction, which can be used to track its status. Each neighboring node also validates the transaction when it receives it and stores a copy of the transaction before propagating it to all their neighbors [36].

A transaction is considered valid if: i) it is encoded as a properly formatted Recursive Length Prefix (RLP); ii) its signature is valid; iii) the transaction’s nonce is equal to the sender account’s nonce; iv) the transaction’s gas limit is higher than the intrinsic gas used by the transaction (a predefined cost of computing the transaction, plus a gas fee for the data sent with the transaction.
Figure 2.3: Lifecycle of a transaction with a contract account as a destination address. Adapted from: [41][42].

and additional gas if it is a contract creation transaction); and v) the sender’s account balance must be able to cover the upfront gas cost (the maximum cost of the transaction, which is calculated by multiplying the transaction’s gas limit by the transaction’s gas price, plus the total value being transferred from the sender to the recipient) [38].

The Ethereum network is composed of both miner and non-mining nodes. Miner nodes maintain a transaction pool to which they add received transactions (step 4 in Figure 2.3), sorted by gas price. The transactions selected to be included in a candidate block are picked from this pool (step 5 in Figure 2.3). Because transactions must be processed sequentially, according to their nonce, if the transaction pool includes two transactions sent by the same EOA, one with the nonce 0 and the other with nonce 2, then the second transaction will be
queued until the transaction with nonce 1 has been seen by the miner [2].

Every transaction included in the block is executed, first by the miner node and then by every other node in the network during block validation. The execution of a transaction starts with incrementing the nonce of the originating account by one and decrementing the balance of the originating account by the part of the transaction’s upfront gas cost [38]. Whether the execution is successful or it fails, the stateRoot is always modified, the transaction is appended to the block’s transaction trie and the transaction receipt is appended to the block’s receipt trie [2].

In Ethereum, the 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 [2]. 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 [36][2][38].

Proof of Work [35][36] is a consensus algorithm that allows nodes in the network to agree on a set of canonical updates to the blockchain state. To successfully generate a new block, miners must find the nonce to include in the block such that the block’s hash starts with a given number of zero bits. This is a computationally expensive operation because it requires the repeated computation of the hash of the block with different nonces. As more blocks are appended to the blockchain, it becomes exceedingly difficult to tamper with a past block as it requires that the attacker redo the PoW for the block and all succeeding blocks as well. The Proof of Work difficulty varies according to the average number of blocks generated per hour. This is to account for increasing hardware speed and varying interest in running nodes, over time. Proof of Work also solves the problem of deciding who can participate in the consensus protocol while simultaneously preventing Sybil attacks [43]. In a permissionless blockchain, anyone is free to participate in the consensus protocol, which makes it possible to forge the identity of processes. In a Sybil attack, a user tries to take over the network by doing just that. Proof of Work makes such attacks impractical because the weight of a single node in the consensus voting process only extends as far as its computation power allows.

When a miner successfully mines a block, it receives a reward. This has a number of purposes [36]. First, it works as an incentive for nodes to participate in the network. Second, it also works as a wealth distribution mechanism, as mining is the only method for generating new coins. And finally, it encourages nodes to stay honest. With rewards, it is more profitable to follow the rules than to try to defraud the system.

Once the miner node finds the PoW that makes the block valid, it adds it to the block header and broadcasts the block to its neighboring nodes (steps 6 and 7 in Figure 2.3). Nodes that
receive the block check that the PoW in the block header is valid, that the state transitions are also valid before propagating the block to their neighboring nodes (steps 8 to 11 in Figure 2.3). Determining that the state transactions are valid, requires the node to [36]:

1. Execute all transactions in the block, in the same order, starting with the initial state equal to the state at the end of the previous block;

2. Add the block reward to the miner’s account;

3. Compare the hash of the final state trie root to the block’s stateRoot. If they are equal, then the state transitions are valid.

Miners are competing among themselves to find the next block in the chain, which means that it is possible for two or more miner nodes to find the next block on the chain at nearly the same time, causing parallel chains to appear. This is called a fork [36][38]. Now, in this situation, blocks with the same level in different parallel chains will probably not contain the same set of transactions, in the same order.

In Ethereum, a fork is eventually resolved for the longest branch. The higher the depth of a block in the blockchain, the smaller it is the probability the block 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’s 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 [11].

**Smart Contracts**

Smart Contracts [2][44] are deterministic computer programs that are executed in the context of the Ethereum Virtual Machine (EVM), typically programmed using a high-level programming language, such as Solidity. Solidity [45] is a statically typed programming language designed specifically to write Smart Contracts. We discuss Solidity in more detail in Section 2.2.5. The contracts are compiled into EVM bytecode and then they are deployed on the blockchain network with a special contract creation transaction [38]. The transaction receipt generated from the execution of a contract creation transaction includes a field that contains the address of the newly deployed contract. Smart Contracts are immutable which means that once a Smart Contract has been deployed it cannot be modified.
There are two modes of Smart Contract execution: calls and transactions [46]. Calls are read-only executions of the Smart Contracts. Because the call is not propagated to the remaining Ethereum nodes and is only executed by the local node it cannot change the blockchain state. Calls also do not consume any gas, which makes them useful for *dry-run* or *what-if* exercises, i.e. to determine what would happen if the same inputs were given to the network. If the destination address of a transaction is a contract account, the contract’s execution is triggered first when the block containing the transaction is mined and later when the peer nodes validate the block. In contrast, transactions executed by all the nodes in the network and modify the blockchain state.

Contract execution is handled by the Ethereum Virtual Machine (EVM), which runs as a local instance on every Ethereum node. Contract execution is handled by the Ethereum Virtual Machine (EVM), which runs as a local instance on every Ethereum node. The EVM runs as a sandboxed copy of the Ethereum state. Upon successful completion of the contract’s execution, the world state is updated to match the sandboxed version. If the execution halts, for any reason, then all changes applied to this sandboxed state are discarded. However, a failed transaction is still recorded as being executed and all the gas spent is charged to the originating account. In both cases, a transaction receipt is generated containing log entries that describe the transaction’s execution (i.e. the gas used by the transaction, the status code of the transaction, the logs created during the execution, etc.) [2].

During the code’s execution, the gas supply is decremented according to the cost of each operation executed. If at any point, the gas supply is exhausted, the execution halts immediately. This makes the EVM a quasi-complete Turing machine because all executions are limited to a finite number of computational steps by the gas supply, avoiding a situation where an execution might run forever, thus bringing the whole Ethereum system to a halt [2][38][44].

After the transaction’s execution completes, the remaining gas, which was not used by the transaction, is refunded to the originating account. The ether paid by the originating account for the gas spent on the transaction represents a transaction fee that is given to the miner that successfully mines a block with the transaction[38].

A contract can invoke another contract through a *message call* [38]. With each *message call*, another EVM is instantiated with the state from the sandbox of the EVM at the level above [2]. Independently of how many contract invocations a transaction may generate, the outer transaction (which may comprise multiple inner invocations) is atomic. Atomicity [33] 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.

An invocation of a Smart Contract by another Smart Contract generates what we refer to as *internal transactions*. Internal transactions can modify the contract state but are not explicitly stored in the blockchain.

### 2.2.4 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. However, a *fork* might cause speculation to fail.

Figure 2.4 shows an example where two parallel chains are formed, each with a different ordering of transactions. Two transactions T1 and T2 invoke the same contract function with input 15 and input 6, respectively. The function’s execution is successful if the contract’s variable \(x\) has a value greater than the input, and decrements the value of \(x\) by the input value. Otherwise, the execution will fail, and the value of \(x\) does not change. In the parallel chain \(a\), transaction T1 is successfully executed first and the transaction T2 is executed at a later time and it fails. The final value of variable \(x\) in the chain \(a\) is 5. In the parallel chain \(b\), the order of the two transactions is reversed and T2 succeeds while T1 fails. The final value of the variable \(x\) in chain \(b\) is 14. T1 and T2 are an example of non-commutable transactions which, if executed in a different order, have a different outcome. In contrast, transactions T3 and T4 are commutable. They both invoke the same contract function which adds an element to a set. Regardless of the order in which they are executed, the set has the same contents after both complete.

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, com-
mutable operations can be executed with more relaxed consistency, (i.e. waiting for fewer con-
firmations) and thus are faster. Programmers need to be aware of the consistency requirements
for each operation to be able to optimize application performance.

Nevertheless, forks affect only less than 10% of Ethereum blocks, and 56% of the concurrent
blocks generated with for the same depth are identical (i.e. have the same transaction set) [17],
hence the speculation will be correct most of the time.

### 2.2.5 Ethereum Stack

DApps are blockchain-based decentralized applications, that allow users to access web applica-
tions to interact with Smart Contracts in the blockchain [10]. Figure 2.5 illustrates the typical
technology stack used by DApps using Ethereum. Smart contracts, implemented in a high-level
language such as Solidity, 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 [14], provided by the Ethereum blockchain, or a high-level interface, e.g. Web3.js [15].
Additionally, it might also use the subscription feature supported by Web3.js to monitor updates
to the blockchain state.

Currently, developers are faced with a shortage of tools to aid third-party development and
community support [47]. Moreover, current development tools present significant limitations
[48][47]. Solidity has a lack of general-purpose libraries, safety checks and memory manage-
ment support. There is also a shortage of tools for verifying the code correctness and powerful
debuggers. Compounded by the fact that, once deployed, the Smart Contract source code is
immutable and publicly visible, which means that anyone can try to exploit it, as was the case
with the DAO attack [49]. In brief, implementing correct Smart Contracts very challenging
endeavor.

Developers struggle with handling gas problems and face EVM limitations (e.g., inefficient
code execution and limited stack size). Furthermore, developers are confronted with the lack of
reference code, up-to-date documentation and the difficulty in finding other developers to review
their code.

**Solidity**

Solidity [45] is a procedural language with similar syntax to JavaScript. Contracts in Solidity
are similar to classes in object-oriented languages, containing persistent data in state variables
and functions which manipulate these variables. Solidity variables types include: \texttt{uint}, \texttt{int}, \texttt{byte}, \texttt{bool}, \texttt{string}, \texttt{address}, \texttt{mapping}, \texttt{enum}, and \texttt{struct}. Automatic getters are generated for every \texttt{public} state variable defined in the contract.

Smart contracts have access to a set of globally defined variables and functions that can be used to obtain information about the blockchain. The \texttt{msg} object represents the transaction or the message call which triggered the contract’s execution and contains the following attributes: \texttt{sender}, \texttt{gas}, \texttt{data}, and \texttt{sig}. The \texttt{tx} object represents the transaction context and contains the attributes: \texttt{gasprice} and \texttt{origin}. The \texttt{block} object contains information about the current block: \texttt{blockhash}, \texttt{blocknumber}, \texttt{blocktimestamp}, etc. The \texttt{blockhash(uint blockNumber)} function returns the hash of the block at the given depth.

In Solidity, a function’s syntax is the following:

\begin{verbatim}
function FunctionName([parameters]) {public|private|internal|external} [pure|constant|view|payable] [modifiers] [returns (return types)]
\end{verbatim}

The keywords \texttt{view/constant} and \texttt{pure} guarantee that the function will not modify the contract’s state, even if executed in the context of a transaction. A \texttt{constant} function will read the state but not change it. The keyword \texttt{view} is an alias for the keyword \texttt{constant} that will be deprecated in a future release. \textit{Getters} are typically constant functions. A Contract\texttt{pure} function is one that is solely dependent on the input parameters, such as a utility function, and that promises to not read or make any changes to the blockchain’s world state. Constant and pure functions should usually be executed in the context of a call. If invoked in the context of transactions they will not change the contract’s state but will still cost gas, as the transaction
has to be executed by every node in the network.

Solidity allows us 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.

In Solidity, events are high-level objects that allow Smart Contracts to interact with the logs entries in the transaction receipt. When a Smart Contract emits an event, the arguments passed to the event are stored in the transaction receipt logs. They provide a useful tool that allows client applications to monitor specific actions that occur during the contract’s execution. At most three of the events arguments can be indexed, but events can also have non-index arguments. It is possible to filter for an application to filter events by specific values of indexed arguments.

**Ethereum JSON RPC API**

Ethereum provides a JSON RPC API [14] to interact with the Ethereum blockchain and with the Smart Contracts. The two most important methods provided by the JSON RPC API are the `eth_sendTransaction` and the `eth_call` methods.

The `eth_sendTransaction` method is used to submit transactions to the Ethereum network. It receives as input the transaction object, which contains the transaction’s request vectors, as described in Section 2.2.1.

Only if the transaction’s destination address is a contract account is the data parameter of the transaction object be defined. If defined, it contains the function selector and the function arguments encoded according to the contract’s application binary interface (ABI). This method returns the transaction hash once the transaction has been propagated to peer nodes. The transaction hash can then be used to track the transaction in the blockchain.

The `eth_call` method triggers a local, read-only execution of a contract function. It receives as input: i) the call object, which is similar to the transaction object except it does not contain a nonce, and ii) an optional the block identifier, which can either be a block number or a tag ("latest", "earliest" or "pending").

If the block identifier is omitted, the contract will execute on the most recent blockchain state the node has seen. Otherwise, it will execute on the blockchain state stored at the specified block. This method returns the return value of the executed contract function. One of the transaction’s request vectors is the transaction nonce. Recall that the nonce must be different for every transaction submitted by the same client account and the transactions by the same client are processed in the order given by the nonce. If the transactions with smaller nonces have
not yet been mined, then the older transactions will remain in the transaction pool and cannot
be included in a block by any miner. To determine the correct nonce for the new transaction, we
can use the method `eth.getTransactionCount`. This method receives as input: i) the account
address, and ii) the block identifier.

This method returns the number of transactions sent from the given account considering
only the transactions included in the block at least as deep as the identified block.

The nonces start at zero and therefore, usually the nonce of the new transaction should
be the number of sent transactions. However, if multiple processes are trying to submit new
transactions from the same account concurrently, we can identify a race condition. This is
because it is possible that both processes will reach the same nonce value. If two transactions
are submitted with the same nonce, then only the transaction which is mined first will be
executed. The other transaction will be discarded. Programmers must be conscious of this
possibility and track the transaction until it is committed in the blockchain to ensure that it
was not discarded because another transaction with the same nonce was ordered before it.

Note that the `eth.sendTransaction` method does not return the value of the Smart Con-
tract’s execution. The most common method to obtain the execution’s return value is through
events. A programmer can achieve this by programming the Smart Contract function to emit
an event (see Section 2.2.5) with the return values. The events are stored in the transaction
receipt logs. Figure 2.6 shows the contents of a transaction receipt which contains a log entry
corresponding to a transfer event. The first item of the topic field, highlighted in the figure,
contains the encoded event signature, and the remaining two items are the encoded indexed
event arguments. The data field, which is also highlighted in the figure, contains the encoded
non-index event parameters.

```solidity
event Transfer(address indexed _from, address indexed _to, uint256 _value)
```

The transaction receipt object (Figure 2.6) can be obtained using the method
`getTransactionReceipt`, which receives as input the transaction’s hash.

The `eth.call` method provides the simplest method for reading the contract state, but we
can also read the value of a contract state variable with the method `eth.getStorageAt`. This
method receives as input i) the address of the Smart Contract account, ii) the index, which is
the position of the Smart Contract storage to read, and iii) an optional block identifier.

Similarly to the `eth.call` method, if the block identifier is omitted, the state is read from
the most recent block in the chain, otherwise it is read from the specified block. Each index
corresponds to 32 bytes in the Smart Contract storage. This means that to read a state variable
that occupies more than 32 bytes of the contract’s storage, we need to do multiple invocations.
Also, it is not trivial to determine the index of each contract state variable. To read the contract’s state at a specific block depth, which as we saw in Section 2.2.1 relates to a specific probability of the value’s consistency, a programmer must proceed as follow: first he must obtain the identifier of the block at the given depth. To obtain the identifier of the block at the head of the chain he can use the method `eth.getBlockNumber`. This method receives no input parameters and returns the identifier of the current block. Since the block’s identifiers are attributed sequentially, he can deduce the identifier of the block at a certain depth by decrementing the current block identifier by the depth. However, because this process is not atomic, and by the time he deduces the identifier, new blocks might already be mined, he might be reading older data than expected. All of this is complicated and error-prone as it requires that the programmer mix the application logic with these operations.

Web3.js

Web3.js [15] is a collection of JavaScript libraries that provides methods for interacting with the Ethereum blockchain and Smart Contracts. There are also Web3 implementations of back-end APIs for multiple languages, such as Python, Java, Kotlin, and PHP, among others.
// Create Contract instance from ABI
const contractABI = [...]  
const contract = new web3.eth.Contract(abi, contractAddress)

// Transfer some tokens
web3.eth.getTransactionCount(account1, (err, txCount) => {
const txObject = {
  nonce: web3.utils.toHex(txCount),
  gasLimit: web3.utils.toHex(800000),
  gasPrice: web3.utils.toHex(web3.utils.toWei('10', 'gwei')),
  to: contractAddress,
  data: contract.methods.transfer(account2, 1000).encodeABI()
}

const tx = new Tx(txObject)
const serializedTx = tx.serialize()
web3.eth.sendSignedTransaction('0x' + serializedTx.toString('hex')).on('confirmation', function(confirmationNumber, receipt){
  if(confirmationNumber == 0){
    console.log("Transaction was\n mined:", receipt.transactionHash)
  }else if(confirmationNumber == 12){
    console.log("Transaction was\n committed:", receipt.transactionHash)
  }else{
    console.log(confirmationNumber, "", receipt.transactionHash);
  }
})
})

Listing 2.1: Snippet of a JavaScript application sending a transaction invoking a contract function. Adapted from: [52].

The sendTransaction method implemented in the web3.eth package returns a promised combined event emitter (PromiEvent). A PromiEvent works like a Promise but with the additional on, once and off functions which allow developers to define listeners for specific events of a transaction, namely:

- **transactionHash**, which is fired after the transaction has been sent to the network and the transaction’s hash is available;

- **receipt**, which is fired when the transaction has been included in a mined block, and the receipt is available;
• **confirmation**, which is fired every time the transaction receives a confirmation, until the transaction reaches a pre-configured number of confirmations (by default 24)

• **error**, which is fired when an error occurs during the sending of the transaction, or if an out-of-gas exception is triggered.

The Web3.js library provides a **Contract** object, which simplifies the interaction with the Smart Contracts. To create a **Contract** instance, we need to provide the contract’s ABI. Listing 2.1 shows a code snippet of a JavaScript application using the Web3.js library to submit a transaction that invokes a contract function **transfer** (line 19) and defines a listener for the confirmation event generated when the block containing the transaction moves to deeper in the chain (lines 20-28).

The Listing 2.2 shows a code snippet of a JavaScript application using the Web3.js library to call a contract function **totalSupply** which receives no arguments and returns an unsigned integer.

```javascript
1 /* Create a contract instance */
2 const contract = new web3.eth.Contract(abi, address)
3
4 /* Call the contract method totalSupply */
5 contract.methods.totalSupply().call((err, result) => { console.log(result) })
```

Listing 2.2: Snippet of a JavaScript application reading data from a contract. *Source: [52].*

The **subscribe** method implemented in the **web3.eth** package allows subscribing to specific events, such as:

1. **newBlockHeaders**, which fires an event for each incoming block header, allowing us to track changes to blockchain;

2. **pendingTransactions**, which fires an event for each incoming pending transaction.

### 2.3 Related work

We now present some works which try to summarize the challenges, proposed improvements and advancements of Smart Contracts [48][47].

Past reports highlight a wide range of vulnerabilities that affect the security of Smart Contracts [53] e.g. reentrancy attack, integer overflow and underflow, transaction dependence order, timestamp dependency, etc.
Sergey and Hobor [54] have proposed that accounts using Smart Contracts are like threads using concurrent objects, with internal mutable state, which manage resources. Without utilizing proper synchronization, Smart Contracts can manifest the same racing conditions as threads, leading to corrupted data. Recent works propose frameworks such as Maticore [55] and Zeus [56], which rely on symbolic execution to find bugs and verify the code of real-world Smart Contracts. Rodler et al. [57] present Sereum, a tool to accurately detect and prevent reentrancy attacks in existing, deployed contracts. Albert et al. [58] describe how reentrant calls complicate smart contract reasoning and may lead to reentrancy attacks. The authors use static analysis to prove a Smart Contract is immune to reentrancy. While these works address correctness at the Smart Contract level, Speculatra targets the broader DApp ecosystem and be used as a complement to these approaches. Another research avenue has proposed algorithms that address the lack of privacy-preserving mechanisms in blockchains, as most transaction data is publicly visible. Cheng et al. [59] present Ekiden, which combines blockchains with Trusted Executed Environments to enable confidentiality-preserving Smart Contracts and high scalability. These works, among others, focus on security and privacy concerns of DApps, whereas Speculatra focuses on reducing the latency of DApps and abstracting the complexities of the Smart Contract state.

A relevant body of research has tried to improve the high transaction commit latencies of Smart Contracts in permissionless blockchains. Dickerson et. al [60] have adapted techniques from software transactional memory (STM) to allow miners to execute non-conflicting contracts in parallel and discover a serializable concurrent schedule for transactions, thereby increasing transaction throughput. Anjana et. al [61] also developed a framework to enable miners to concurrently execute Smart Contract transactions using STM.

Lu and Peng [62] have proposed a blockchain processing unit (BPU), for accelerating smart contract execution using an FPGA. These works focus on speeding up the smart contract execution. Speculatra focuses on a different layer of the development stack, in which the implementation details of the mechanisms are abstracted, and focuses on how DApps interact with Smart Contracts.

Cook et. al [63] presented Hash-Mark-Set(HMS), for increasing smart contract transaction throughput by exposing a read-uncommitted view of the smart contract variables and information about transaction dependencies to miners, so that they can adjust block order. Concurrent applications using a Smart Contract do not need to wait until a block is committed to see a change in a variable. This prevents transactions from failing due to using a stale value (e.g., insufficient funds). The throughput of successful transactions may increase up to a factor of five.
While HMS exposes an early uncommitted view at the smart contract level, Speculatra exposes early speculative values at the DApp level.

Nathan et al. [64] presented the design of a permissioned blockchain relational database on top of PostgreSQL, executing Smart Contracts as stored procedures and guaranteeing serializable snapshot isolation (SSI). El-Hindi et al. [9] presented BlockchainDB, which implements a database layer on top of Smart Contracts and permissioned blockchains, supporting sequential consistency and eventual consistency. While these works provide consistency guarantees at the database level, Speculatra provides increasing consistency guarantees at the DApp level.

2.4 Speculative Execution

Speculative execution is a general computer science concept that has successfully been applied to improve performance of many types of systems.

Steffan et al. [65] present Thread-Level Speculation (TLS), which allows compilers to optimistically create parallel threads in the presence of statically ambiguous data dependences. TLS leverages invalidation-based cache coherence to detect data dependence violations at runtime.

Fraser and Chang [66] present an in-kernel disk prefetcher that takes advantage of spare processing cycles to automate prefetching for applications that stall on disk I/O. While applications are stalled on I/O, it uses speculative execution to try to determine which data the application is likely to require in the near future and initiate prefetching. Based on the assumption that the application speculative pre-execution will be similar to the application’s future non-speculative execution, this solution provides benefits of up to 60% for a wide range of explicit-I/O and swapping applications.

Nightingale et al. [67] presents Speculator, which improves the performance of distributed file systems by hiding latency of I/O operations. Speculator allows multiple process to share speculative state by tracking causal interdependencies between processes. Using Speculator, a file system predicts the result of remote operation and continues execution speculatively, rather than block awaiting for the operation result. If the prediction is false, the system is rolled back to the checkpoint state of the calling process. Otherwise, the checkpoint is discarded.

Wester et al. [68] propose a mechanism that hides latencies in clients of replicated services. Geographic distribution is important for maximum fault tolerance. Unfortunately, it tends to result in high network latencies. Additionally, protocols that tolerate Byzantine faults must wait for multiple replicas to reply. This work allows clients to continue executing speculatively after receiving the first response. By tracking all effects of the speculative execution and hiding the
speculative state, they can undo the effects of a mispeculation.

Kemme et al. [69] proposes a solution that hides the overhead of the atomic broadcast mechanism by overlapping the coordination fase of the atomic broadcast with the transaction execution. The transaction manager uses the order in which transactions arrive at each site to determine a tentative scheduling order. Once the definitive total order is determined, if it matches the tentative order, the transactions are committed. Otherwise, steps are taken to ensure the execution order obeys the definitive total order, which might involve aborting and rescheduling transactions.

Reddy and Kitsuregawa [70] proposes speculative locking (SL) protocols that increase parallelism among conflicting transactions without violating serializability criteria. SL allows transactions to access after-images produced during the execution of lock-holding transactions. By doing so, the waiting transaction can carry out speculative executions and retains execution based on the final result of preceding transactions. Additionally, this work also proposes SL variants that can significantly reduce the number of speculative executions, which under the naive version of SL explode with data contention.

Traditional distributed machine learning (ML) systems typically employ an Asynchronous Parallel (ASP) model, where works proceed to the next iteration before receiving the latest model parameters. This model sacrifices training quality in favor of decreased synchronization overhead. Zhang et al. [71] proposes speculative synchronization, a mechanism that enables computation to speculate on the recent parameters and, uses an heuristic algorithm to determine when a training iteration should be restarted with fresher parameters. This mechanism can achieve up to 3x speedup over the tradition asynchronous parallel scheme.

Zyzzyva [72] leverages speculation to reduce the cost of BFT machine replication. In Zyzzyva, replicas optimistically adopt the order proposed by the primary and respond immediately to the client. Clients detect inconsistencies and help replicas converge on a single total ordering of requests.

Speculative execution is a well research technique, which has been successfully applied to reduce latencies, to a large set of areas. As such, its application is very promising in the context of Speculatra.

2.5 Discussion

Considering that the last few blocks of a blockchain are susceptible to change, due to the occurrence of forks, we might be lead to believe that blockchains can only support eventual consistency. However, this statement is not precise. Depending on how we handle concurrent
read and write operations, we can support different consistency models. If an application chooses to only read the state from committed blocks and ensures that it always reads a state that reflects all the transactions it has submitted, we can guarantee sequential consistency. This, of course, has a severe impact in the application’s latency due to the high commit times displayed by blockchains. On the other end, if an application chooses to read the state from the most recent block in the blockchain, then it foregoes strong consistency in favor of better performance.

Speculative execution has been successfully applied to many use cases to reduce latencies. Blockchains also seem a good candidates for optimizing performance through speculation as the high commit times result in a significant window for speculative execution and the low probability of forks mean that we can expect the speculation to be successful most of the time.

Moreover, the current programming model is very complex and error-prone. Additionally, the blockchain landscape is still currently in its infancy and far from widespread adoption by the developer community. As a result, there are few skilled blockchain application developers. A developer with an average knowledge of blockchains will likely either choose to adopt: i) the more conservative approach, which fails to fulfill the potential performance of the DApp; or ii) a relaxed approach, which might dangerously impact the correctness of critical DApp operations.

Previous works focus on improving the development of Smart Contracts and the EVM performance. However, they fail to address the challenges and limitations of the current interfaces for interacting with Smart Contracts. With this work, we address this gap left in current literature.
Chapter 3

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 3.1.

To add a product P to the cart, a customer must submit a transaction T1 invoking the addToCart function (line 25-34). 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 38), which returns the current list of products in the cart and the 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 y 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 53) after seeing the speculative contents of the cart, the customer would ultimately purchase a different set of products than he had believed.

To avoid this, we have several alternatives. The most conservative approach requires waiting that all transactions which affect the product’s stock or update the customer’s cart be committed before issuing the transaction T2. While this approach ensures the correctness of the operation, it presents a considerably large latency.
Table 3.1: Comparison of three different approaches for issuing transactions based on the Smart Contract state.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Latency</th>
<th>Gas consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conservative approach</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Compensating transactions</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Speculative transactions</td>
<td>Low</td>
<td>Medium</td>
</tr>
</tbody>
</table>

A second alternative would be to issue T2 based on the speculative view of the cart contents. If speculation later fails, a **compensatory transaction** T3 must be issued reverting the effects of T2. At first glance, this alternative seems advantageous considering that a mispeculation will occur less than 10% of the time (as only 10% of blocks are affected by forks). However, it is not without issues. First, it requires issuing at least two transactions T2 and T3, which have a non-negligible cost, even if the customer ultimately does not confirm the purchase. Additionally, the blockchain state will also be temporarily inconsistent as the results of T2 are exposed before T3 is executed.

A third alternative relies on **speculative transactions**. To support speculative transactions, we must add 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. As a result, 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, we need to consider that preconditions for some use cases might not be obvious when the Smart Contract is implemented, and once deployed, the Smart Contract code is immutable. Therefore, 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 do not need to wait for the transaction to be committed. It is also less expensive than the second approach 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.

Table 3.1 summarizes the three approaches we just discussed in terms of latency and gas consumption. In the next chapter, we introduce the design of Speculatra.
pragma solidity ^0.4.10;

contract Store {
  mapping (address => Customer) customers;
  mapping (uint256 => Product) products;
  uint256 private store_balance;

  struct Customer {
    address addr;
    bytes32 name;
    Cart cart;
    uint256 balance;
  }

  struct Cart {
    uint256[] products;
    uint256 totalSum;
  }

  struct Product {
    uint256 id;
    bytes32 name;
    uint256 units;
    uint256 price;
  }

  /*Add a product to the customer cart if an unit is available*/
  function addToCart(uint256 id) returns (bool success) {
    Customer customer = customers[msg.sender];
    Product prod = products[id];
    // If product has available units, add to cart
    if (prod.units > 0) {
      customer.cart.products.push(prod.id);
      return true;
    }
    return false;
  }

  /*Returns the list of products in the customer cart and its total price*/
  function getCart() constant returns (uint256[] memory product_ids, uint256 total) {
    Customer customer = customers[msg.sender];
    uint256 len = customer.cart.products.length;
    uint256[] memory ids = new uint256[](len);
    customer.cart.totalSum = 0;
    for (uint256 i = 0; i < len; i++) {
      ids[i] = products[i].id;
      customer.cart.totalSum = total + products[i].price;
    }
    return (ids, customer.cart.totalSum);
  }

  /*Checkout process that’ll use the current shopping cart to transfer balances between the customer and the store*/
  function checkoutCart() returns (bool success) {
    Customer customer = customers[msg.sender];
    uint256 totalSum = customer.cart.totalSum;
    if ((customer.balance >= totalSum) &&
      customer.cart.products.length > 0) {
      customer.balance -= totalSum;
      customer.cart = Cart(new uint256[](0), 0);
      store_balance += totalSum;
      return true;
    }
    return false;
  }
}

Listing 3.1: Solidity Smart Contract which implements some basic functionality of an eCommerce DApp.
Chapter 4

Speculatra

In this chapter, we present 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.

Figure 4.1 shows the technology stack of a DApp that uses Speculatra. In the traditional technology stack (Figure 2.5), DApps read and update the Smart Contract’s state using the Web3.js [45], or a similar interface. Speculatra DApps still use these same interfaces to issue updates, but use Specultra’s API, instead, to read the blockchain state. In both stacks, the Smart Contracts are implemented in a high-level language, such as Solidity [45], however, to support speculative transactions, Speculatra Smart Contracts must be implemented according to a specific framework, which we present in Section 4.6.

We start by an high-level overview of the Speculatra API, and how it can be used by applications (see Section 3), before discussing how this is materialized. The main method offered by Speculatra is:

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

This method allows developers to speculate on the result of uncommitted transactions. Its arguments are: i) `data`, the transaction object for the method to be called; ii) `speculateCallback`, the callback function that will process the speculative result; iii) `abortCallback`, the callback function that will handle errors and failed speculations and iv) `commitCallback`, the callback function that handles the speculation confirmation. It returns a `call object`, which we describe in Section 4.2.
An Example

Listing 4.1 shows an example of a DApp using this interface. The DApp initializes Speculatra (line 2), specifying the address of Smart Contract defined in Listing 3.1. 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—21). Otherwise, once the speculation is confirmed, the purchase is also confirmed (lines 22—23).

4.1 Requirements

The API introduced before should observe a set of requirements, which we identify below:

1. **Ensure sequential consistency.** To ensure sequential consistency, reads must always ultimately return the committed state. Additionally, to ensure that the read-your-writes (RYW) property is not violated, reads must also reflect the effects of all transactions issued by the application during its execution.

2. **Leverage speculation to reduce the latency of operations.** Speculatra must al-
var contract = new Contract(jsonInterface, address);
var speculatra = Speculatra(address);

function purchase(whishlist){
  whishlist.array.forEach(product => {
    // Issue txs adding each product to the cart
    contract.methods.addToCart().send({from:...});
  });
  speculatra.speculate(
    contract.methods.getCart(),
    speculate=function(res){
      var cartContents = res[0];
      // If the cart contains the same set of products
      // issue a tx finalizing the purchase
      // but do not confirm the purchase yet
      if (whishlist.equals(cartContents)) {
        contract.methods.checkoutCart().send({from:...});
      }
    },
    abort=function(res, error){
      abort(error);
    },
    confirm=function(){
      // Confirm the purchase when the speculation is confirmed
      confirm();
    }
  );
}

Listing 4.1: Basic eCommerce DApp using Speculatra.

...low applications to overlap the commit waiting period with further computation, hence allowing to reduce the perceived latency.

3. **Maximize freshness.** By maximizing freshness, Speculatra also reduces the probability of conflicts in the speculative period.

4. **Compatibility with the current Ethereum implementation.** To ensure compatibility with the current Ethereum implementation, Speculatra must not require any modification to the current Ethereum protocols, to the EVM nor Smart Contract programming languages. Additionally, it should require as little changes as possible to existing Smart Contracts so that it can easily be adopted by existing blockchain applications.

5. **Abstract internal details and behaviors of the Ethereum blockchain.** This will allow developers to separate application logic from the Smart Contract state management logic and lead to simpler code, and hence less prone to bugs and mistakes.

To fulfill the requirements we just listed, 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 have not been executed. Additionally, to increase the probability of successful speculation, we want to read from the most consistent state possible.

To maximize freshness we also need to consider concurrent transactions updating the contract state, 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.

To ensure sequential consistency, Speculatra must be able to reliably detect failed speculations. Intuitively this might seem simple, as mispeculations are always a result of a fork, which are fairly easy to detect. However, as discussed in Section 2.2.4, not all forks will cause speculation to fail. The fork might not affect the set of transactions that modify the contract state, which we will refer to as the transactions relevant for the speculation from now on, either because they are not included in a block discarded by the fork or they appear in the relative same order in the new blocks. Conversely, even if these transactions were affected by the fork, the transaction may be commutable, which means the contract state remains the same.

As discussed in Section 3, 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 transparent abstraction adequate for this scenario.

Finally, we choose to implement Speculatra in JavaScript because Web3.js [15] is the most popular library for interacting with the Ethereum blockchain. However, JavaScript is single-threaded, which is not ideal for asynchronous communication.

4.2 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\}
- **blockNumber** – height of the block containing the transaction
- **blockHash** – hash of the block containing 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 removed 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, possibly empty.
- **state ∈ {PENDING, EXECUTED, VERIFY}**
- **result** – stores the speculative result, but initially is **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**, to distinguish them from tracked transactions, i.e. the complete set of transactions monitored by Speculatra.

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**.

Figure 4.2 shows the state diagram of a transaction in Speculatra. When a transaction is flagged as a potential update of the contract state, it is added to the list of tracked transactions,
with the state pending. Once the transaction is included in a mined block, its state is changed to uncommitted and finally to committed once it has received 12 confirmations. The commit callback is also called at this stage. If the transaction is discarded by the local Ethereum node, the discard callback is called and the transaction is removed from the list of tracked transactions. The same occurs if the transaction is not mined within two minutes\(^1\), to avoid the whole system stagnating due to a single transaction. In fact, all speculation requests submitted after a transaction T1 is registered depend on transaction T1, hence the respective calls can only be executed once the transaction T1 has been mined. Therefore, if the transaction is neither mined nor considered discarded, the system stops making progress.

Figure 4.3 shows the state diagram of a call. Initially, the call state is set to pending. Once the transactions on which the call depends have been included in a block in the blockchain, the call is executed. If the execution is successful, its state is updated to executed and the speculate callback is triggered with the speculative value. Conversely, if the execution fails, the abort callback is triggered with an error message indicating why it failed. If we suspect that the speculation has failed, the call’s state is changed to verify. If we confirm that the speculation has failed, the call’s state is updated to failed and the abort callback is triggered with an error message indicating that the speculation failed and the value after the reorganization. Otherwise, the call’s state is once again set as executed. Once all the transaction on which the call depends have been committed, the speculation is confirmed and the confirmation callback is triggered.

### 4.3 Tracked Transactions

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.

\(^1\)The 2 minutes is the default timeout used by Web3.js for waiting for a transaction to be mined.
As discussed in Section 2.2, transactions can interact with Smart Contracts directly, or indirectly by executing a contract function that invokes another contract. In the first case, the transaction’s destination address is the Smart Contract address and the transaction’s data indicates which Smart Contract function is invoked. In the second case, however, this is not as simple. An indirect invocation of a Smart Contract is not registered in the blockchain. As a 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.

First, it 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 [73] 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 4.5.

One possible solution for this second issue would be to filter this set of transactions by, iterating over the most recent blocks in the blockchain, starting from the block in which the most recent transaction is included and moving backward until we reach the block containing the oldest uncommitted transaction. For each block, we would compare the smart contract state with the smart contract state in the previous block. This comparison can be achieved by simply comparing the Smart Contract’s storage root (see Section 2.2.2). If the blockchain state does not change from one block to the next, then the transactions included in the second block do not modify the blockchain state. This comparison can only determine if the transaction corresponds
to a smart contract update if the transaction’s execution was successful. If the transaction failed, the potential changes that would result from its successful execution were discarded. Were the transaction to succeed after a fork occurred, the smart contract state could be altered. We did not implement this solution because we conjecture that it will be too expensive and will not scale well for an increasing number of transactions. This could be further explored in future work.

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, Speculatra considers all transactions which interact with the Smart Contract, issued by any client. Transactions are identified with the method we described above.

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. This method allows developers to specify callbacks that are triggered when the transaction is committed or discarded.

Table 4.1 summarizes the three variants, comparing them in terms of the code complexity of the DApps, the overhead of tracking transactions each presents, the freshness of reads and the read latency for each variant. Variants multiple-writers and selective-writers result in simple application code when compared to variant application-specific since, in the last, the application must register each transaction explicitly. The overhead of tracking transactions will obviously be higher for the variants that track more transactions, while the freshness, and consequently the probability of a conflict, will be the opposite. Finally, the application-specific variant is expected to present the smallest read latency since it considers fewer transactions and therefore it is less likely for a call to be blocked waiting for the dependent transactions to be included in blocks in the blockchain. We evaluate these scenarios in Section 5.3.2.

For the remainder of the document, we will consider the multiple-writers variant as the default.
Table 4.1: Comparison of three variants supported by Speculatra, which differ in how identify tracked transactions.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Code complexity</th>
<th>Overhead</th>
<th>Freshness</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple-writers</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>selective-writers</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>application-specific</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Algorithm 1 Execute call

1: **Globals:** history $\leftarrow$ contains most recent blockchain blocks

2: **Require:** for all $tx \in \text{call.txs}$: $tx.$state $\neq$ PENDING

3: **function** EXECUTECALL(call)

4: callBlock $\leftarrow$ SELECTBLOCK(call)

5: error, result $\leftarrow$ await web3.call(call.data, callBlock)

6: **if** error **then**

7: call.abort(error) $\triangleright$ Call failed

8: calls = calls $\setminus$ call

9: **else if** result $\neq$ call.result **then** $\triangleright$ Speculation failed

10: call.abort(result)

11: calls = calls $\setminus$ call

12: **else** $\triangleright$ Speculation did not fail

13: call.state = EXECUTED

14: call.result = result

15: **end if**

16: **end function**

17: **Globals:** history $\leftarrow$ contains the block history

18: **function** SELECTBLOCK(call)

19: **if** call.txs == $\emptyset$ **then** $\triangleright$ Committed block

20: callBlock $\leftarrow$ history.latest.blockNumber - CONFIRMATIONS

21: **else** $\triangleright$ Block containing most recent tx

22: callBlock $\leftarrow$ MAX(call.txs, $\lambda$ key: $tx.$blockNumber).blockNumber

23: **end if**

24: **end function**

4.4 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 considering 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 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).

Figure 4.4 illustrates which block is selected for different use cases. Consider four transactions T1, T2, T3 and T4, which are included in blocks B1, B2, B3 and B4 respectively, and a fifth transaction T5, which has yet to be included in a block. Transaction T1 invokes the Smart Contract A. Transactions T2 and T3 invoke Smart Contract E and transaction T4 invokes Smart Contract C. Finally, transaction T5 invokes Smart Contract D. Additionally, Smart Contract C invokes Smart Contract E.

To speculate on the state of Smart Contract A, the corresponding call will only be dependent on T1. Therefore, block B1 is selected. Conversely, if we want to speculate on the state of Smart Contract E, the corresponding call will be dependent on transactions T2, T3 and T4. Therefore, block B4 will be selected. To speculate on Smart Contract C, the corresponding call is dependent only on transaction T4, therefore block B4 will be selected. Finally, to speculate on the state of Smart Contract D, the corresponding call is dependent on T5, which has not yet been executed. Therefore, the call can only be executed once transaction T5 is included in a block B5 and the block selected will be B5.

4.5 Detecting mispeculations

Speculatra must be able to consistently detect mispeculations. 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 were not discarded, the call could not have been affected. Addi-
tionally, 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, ii) if so, 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. Figure 4.5 illustrates one scenario in which this can happen. Let us assume that *call A* depends on transactions *T1, T2* but not on *T3*, and that initially all three transactions were included in the same block *B1*, with *T3* executed after *T1* and *T2*. Because we cannot see the intermediate state of a block, when we first execute the *call*, the speculative result obtained also reflects *T3*. Let us now assume that a fork has occurred, and block *B1* was discarded and the transactions *T1, T2* were included in block *B1’* and transaction *T3* was included in block *B2’*. Now, as we repeat the *call A* (line 3, Algorithm 1), the result obtained will not reflect *T3* and will differ from the result of the previous execution, even though the set of transactions included in the blockchain and their ordering remains the same. In this case, we will flag the speculation has failed, even though that is not correct.

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 ← txs with
   tx.state = UNCOMMITTED
   calls ← calls submitted
   history ← stack most recent blockchain blocks

2: **upon event** < reorganization, newBlock > do
3:   discardedBlocks ← UPDATEBLOCKHISTORY(history, newBlock)
4:   affectedTx ← \{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 &
       call.txs ∩ affectedTxs ≠ ∅
       then
10:        call.state ← VERIFY
11:        end if
12:    end for

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

4.6 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. In Section 3 we discussed multiple approaches for speculative blockchain transactions. 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.

The most obvious precondition is to compare the return value of the speculative call with the current return value of the respective contract function. However, while a smart contract
function can invoke another smart contract function, Solidity does not allow us to express on which block’s state we wish to execute the contract function. Rather, it is always executed in the current state of the blockchain. Even if this was possible with Solidity, this would not be sufficient if the transaction was dependent on more than one speculation.

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 limitations. 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, and the applications needs to issue a new transaction to execute the same operation.

Listing 4.2 contains the definition of a Solidity modifier that implements this precondition. It uses the special function blockHash provided by Solidity (see Section 2.2.5) which returns the blockchain for the block with the given block number.

Listing 4.3 shows an example of a Speculatra variant of the Solidity Smart Contract in Listing 3.1. Each contract function that updates the smart contract state is decorated with the modifier defined in Listing 4.2. Besides the arguments of the original function, these functions receive two additional arguments, blockHash and blockNumber.

Because the source code of a Smart Contract cannot be modified once the Smart Contract is deployed, the previous approach does not work for existing Smart Contracts. Listing 4.4 shows an example of a Speculatra wrapper of the Solidity Smart Contract in Listing 3.1. For each contract function that updates the smart contract state, we defined a wrapper method modified with the modifier defined in Smart Contract in Listing 4.2, which received two additional arguments, besides the arguments of the original function, blockHash and blockNumber and invokes the corresponding smart contract method in the original contract.

Listing 4.5 shows how the eCommerce application implemented in Listing 4.1 can be modified to use this solution.

4.7 Implementation

In this section, we describe the implementation details of Speculatra. Speculatra uses the Web3.js library (see Section 2.2.5) to interact with the blockchain.

Upon initialization, Speculatra subscribes to the pendingTransactions to be notified and
all incoming transactions in the blockchain. Once it receives a notification for a new transaction, it requests the corresponding transaction object and analyzes the transaction request vectors. In the *multiple-writers* variant, Speculatra looks at the transaction’s destination address, to determine if it potentially updates the state of the Smart Contract. In the *selective-writers* approach, Speculatra additionally looks at the sender’s account address, to determine if it was issued by one of the specified clients. This mechanism is disabled in the *application-specific* variant.

This method only allows us to see the transactions which have been received after Speculatra has been initialized. However, we must also consider the transactions which have already been included in the blockchain but have yet to be *committed*. Therefore, upon initialization, Speculatra also retrieves the transactions included in the *uncommitted* blocks in the blockchain and analyzes them as well. This mechanism is also disabled in the *application-specific* variant.

Speculatra also subscribes to the `newBlockHeaders`, upon initialization, which emits a notification when a new block is included in the blockchain. This subscription is used to update Speculatra’s *block history* and to detect reorganizations, by comparing the *parentHash* of the new block with the *hash* of the most recent block in *block history*.

We observed that the reorganization depth exceeds one block, the `newBlockHeaders` subscription does not always trigger an event for all the new blocks included in the blockchain during a fork. However, it triggers an event for at least the last block included. Therefore, we can not rely solely on this method to get all the new blocks. Instead, we use `getBlock` to recursively get the new blocks, starting from the last block included in the chain, until we reach the first block that was not affected by the fork.

Once a transaction is *flagged*, it is added to list of tracked transactions. Speculatra monitors the state of *tracked transactions*. When the transactions are *pending*, we periodically poll the transaction state by invoking the `getTransactionReceipt` method, which returns the transaction’s receipt (see Figure 2.6). If the *blockHash* and the *blockNumber* in the receipt are not *null*,

```solidity
class Speculatra {
    modifier verifySpeculation(uint _blockNumber, bytes32 _blockHash) {
        require(
            blockhash(_blockNumber) == _blockHash,
            "Speculation failed."
        );
    }
}
```

Listing 4.2: Solidity Smart Contract that defined a modifier which implements the precondition check that verifies the speculation.
the transaction has been included in a block in the blockchain.

We implemented three different methods for polling the transaction state. In the first method, the transaction state is polled with a fixed periodicity. This is the default method used by Speculatra. In the second method, the polling interval is dynamically adjusted over time according to the current mining rate. When Speculatra receives a new block notification, the polling interval is set to half the average time between \textit{newBlockHeader} events. We choose to halve the average time to account for the variability in the time it takes to mine new blocks, taking into consideration that the delay with which Speculatra detects that a transaction has been included in a mined block directly impacts read latency. Finally, in the third method, the blockchain state is polled when a new block is appended to the blockchain.

The \texttt{newBlockHeaders} subscription is also used to update the number of confirmations for each transaction. Once a transaction reaches 12 confirmations, it is removed from the list of tracked transactions.

4.8 Discussion

Speculatra guarantees sequential consistency by ensuring that all causally-dependent updates are reflected in the read values.

Speculatra is able to reduce the perceived latency of operations by allowing applications to speculate on the outcome of transactions that have not yet been committed, and execute both in-chain and off-chain operations during the \textit{commit waiting} period, hence mitigating the impact of the high \textit{commit times}.

Our implementation of Speculatra required no changes to the Ethereum implementation. Additionally, the only mechanism which depends on the implementation of an application’s Smart Contracts is speculative transactions. However, we presented a method for extending support for speculative transactions to existing Smart Contracts with a \textit{wrapper}.

The read results of the \textit{multiple-writers} variant of Speculatra reflect all updates to the Smart Contract state, thus maximizing \textit{freshness}. Consequently, the probability of a conflict in the speculative transactions due to stale speculative results is minimized.

Finally, Speculatra abstracts the complex details of managing the state of a Smart Contract, such as, identifying which transactions modify the state of a Smart Contract, monitoring the progress of transactions, selecting which block to read from, and detecting mispeculations.
\begin{verbatim}
contract ModifiedStore is speculatra {
    mapping (address => Customer) customers;
    mapping (uint256 => Product) products;
    uint256 private store_balance;

    struct Customer {
        address adr;
        bytes32 name;
        Cart cart;
        uint256 balance;
    }

    struct Cart {
        uint256[] products;
        uint256 totalSum;
    }

    struct Product {
        uint256 id;
        bytes32 name;
        uint256 units;
        uint256 price;
    }

    function addToCart(uint _blockNumber, bytes32 _blockHash, uint256 id)
        public verifySpeculation(_blockNumber, _blockHash)
        returns (bool success) {
        Customer customer = customers[msg.sender];
        Product prod = products[id];
        // If product has available units, add to cart
        if (prod.units > 0) {
            customer.cart.products.push(prod.id);
            return true;
        }
        return false;
    }

    function getCart() constant returns (uint256[] memory product_ids,
                                               uint256 total) {
        Customer customer = customers[msg.sender];
        uint256 len = customer.cart.products.length;
        uint256[] memory ids = new uint256[](len);
        customer.cart.totalSum = 0;
        for (uint256 i = 0; i < len; i++) {
            ids[i] = products[i].id;
            customer.cart.totalSum = total + products[i].price;
        }
        return (ids, customer.cart.totalSum);
    }

    function checkoutCart(uint _blockNumber, bytes32 _blockHash)
        public verifySpeculation(_blockNumber, _blockHash)
        returns (bool success) {
        Customer customer = customers[msg.sender];
        uint256 totalSum = customer.cart.totalSum;
        if ((customer.balance >= totalSum) &&
            customer.cart.products.length > 0) {
            customer.balance -= totalSum;
            customer.cart = Cart(new uint256[](0), 0);
            store_balance += totalSum;
            return true;
        }
        return false;
    }
}
\end{verbatim}

```solidity
contract SpeculatraStore is speculatra {
    // Address of original contract
    address store_addr = 0x889248BA83585dC1d2fB26c8C387A456299fdBa1;

    function addToCart(uint _blockNumber, bytes32 _blockHash, uint256 id)
        public verifySpeculation(_blockNumber, _blockHash)
        returns (bool success) {
        Store store = Store(store_addr);
        return store.addToCart(id);
    }

    function getCart() constant returns (uint256[] memory product_ids, uint256 total) {
        Store store = Store(store_addr);
        return store.getCart();
    }

    function checkoutCart(uint _blockNumber, bytes32 _blockHash)
        public verifySpeculation(_blockNumber, _blockHash)
        returns (bool success) {
        Store store = Store(store_addr);
        return store.checkoutCart();
    }
}
```

Listing 4.4: Solidity Smart Contract wrapper for the Smart Contract in Listing 3.1 which supports speculative transactions.

```javascript
var call = speculatra.speculate(
    contract.methods.getCart(),
    speculate=function(res){
        ...
        contract.methods.checkoutCart(call.blockHash, call.blockNumber)
        .send({from:...});
        ...
    },
    ...
)
```

Listing 4.5: Basic eCommerce DApp using Speculatra to interact with Smart Contract in Listing 4.4.
Chapter 5

Evaluation

In this chapter, we evaluate Speculatra by answering the following questions:

1. 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?

2. What is the computational overhead introduced by Speculatra, compared to solutions that provide weaker consistency guarantees?

In Section 5.1 we start by describing the simulated network we used to evaluate Speculatra in Ethereum. In Section 5.2 we describe the model DApp we considered for Speculatra’s evaluation. In Section 5.3 we evaluate the performance, overhead and scalability of read operations using Speculatra. In Section 5.4 we determine the overhead cost of speculative transactions, supported by Speculatra. Finally, in Section 5.5 we quantify the speedup that can be achieved by Speculatra, if applications take advantage of the speculative period to execute in-chain or off-chain computations. We conclude this chapter with a summary of the evaluation.

5.1 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 [74]. 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) [73]. This required changing \( \approx 300 \) lines of code in Geth and adding another \( \approx 3000 \) lines of code.
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;
  }
}

Listing 5.1: Solidity Smart Contract used for evaluation.

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 [73] (from block 0x50f8f4 up to block 0x5ef20).

We ran each of the experiments described in Sections 5.3.1 and 5.3.2 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 [73].

We used the Linux top command [75] to obtain information about the CPU and memory utilization during the experiments. We collect the average percentage of the CPU time utilized by the process and the average physical memory used by the process.

5.2 Workload

To evaluate Speculatra, we implemented a DApp that interacts with the Smart Contract defined in Listing 5.1, 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. The smart contract functions Add, Subtract, Multiply and Divide receive a single integer argument and apply the corresponding arithmetic operation on Value. Additionally, the smart contract function Clear sets Value to zero.
In every experiment, we assume an execution where the DApp starts by issuing a transaction T1, invoking \texttt{Add(2)}, followed by another transaction T2, invoking \texttt{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 \texttt{boolean}. If the processing result is \texttt{True}, the DApp issues a transaction T3, invoking \texttt{Divide(2)}. Once T3 is confirmed, the execution is complete. Figure 5.1 illustrates the execution process.

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 5.3 to evaluate the first phase of the DApp’s execution. Then, in Section 5.4 we evaluate the cost of issuing transaction T3 before speculative period ends. Finally, in Section 5.5 we evaluate the overall latency of the application’s execution.

Figure 5.1: Flowchart of the execution of the DApp considered for the evaluation.

5.3 Speculative Reads

In this section, we focus on the evaluation of the first phase of the DApp’s execution (see Figure 5.1), and evaluate the latency and overhead of read operations using Speculatra.

5.3.1 Read Latency

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

(A) \textbf{Read-immediately}. Reads as soon as the hash of transactions T1 and T2 return.

(B) \textbf{Read-after-mined}. Reads once transactions T1 and T2 have been mined.

(C) \textbf{Read-after-committed}. Waits for transactions T1 and T2 to receive 12-confirmations, each, before reading.
(D) **Speculatra.** Uses Speculatra, submitting the speculation request once transactions T1 and T2 hashes have been returned.

The first two policies do not guarantee sequential consistency. The first policy violates the *read-your-writes* property as the transactions might not yet be included in a block at the time of the read, and hence, the *call* primitive does not consider their effects when reading. The second policy does not return a value that is consistent with the final order of the transactions if a fork affects the smart contract state after the read. The final two guarantee sequential consistency.

For this part of the evaluation, 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*.

Figure 5.2 illustrates these scenarios. To simulate scenarios *reorder-safe* and *reorder-conflict* we modified one Ethereum node to drop all blocks generated by other nodes in the network, until both transactions were included in the blockchain. Afterward, the node resumes normal functionality. All nodes in the network received the same set of transactions. As a result, the modified node generated a parallel chain to the one generated by the remaining nodes in the network. As the aggregate computational power of the other nodes in the network surpassed the computational power of the modified node, the chain generated by the other nodes was longer and is selected when resolving the fork.

Furthermore, we modified the algorithm that the Ethereum nodes used to select the transactions to include in a block and their order so that, in scenarios *same-order* and *reorder-safe*, all nodes generated similar blocks, with the same set of transactions in the same order, and in scenario *reorder-conflict* nodes generated different blocks, and the final state of the Smart Contract is different in each of the parallel chains.

This required modifying $\approx 50$ lines of code and adding $\approx 100$ lines of code in Geth. All results are the average of 60 runs, each approximately 20 minutes long.

Figure 5.3 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.
(a) Scenario *same-order*

(b) Scenario *reorder-safe*

(c) Scenario *reorder-conflict*

Figure 5.2: Illustration of the scenarios considered for the evaluation described in Section 5.3.1.

until the final read result is returned according to each policy. If the speculation is correct, we consider the time at which the speculative result is returned as the read time of policy *speculatra*. 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, in which case the read time would be longer. We did not evaluate this scenario because, besides being
exceptionally rare, we expect that if the speculation fails \( n \) times, the read time will be \( n \times x \), assuming \( x \) is the read time when speculation fails once.

The results are depicted in Figure 5.3. As expected, the more relaxed policies \textit{read-immediately} and \textit{read-after-mined} present the lowest read times. When comparing policies \textit{read-after-committed} and \textit{speculatra}, which ensures sequential consistency, the policy \textit{speculatra} presents on average lower read times across all the scenarios analyzed. In fact, when the speculation is correct, policy \textit{speculatra} presents only slightly higher read times than policy \textit{read-after-mined} (around 1.30 seconds). When the speculation fails, policy \textit{speculatra} still performs better on average than policy \textit{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\cite{17}, which is smaller than the depth of the reorganization in our simulations (\( \approx 10 \) blocks).

The results for the CPU utilization (Figure 5.3) observed may seem counter-intuitive as the most relaxed policies present the highest values, and policy \textit{speculatra} presents slightly lower values than policy \textit{read-after-committed}. These results can be justified by considering that the total execution time of policies \textit{read-immediately} and \textit{read-after-mined} is considerably smaller than the other two policies, since they terminate as soon as the read returns while the other the
sequentially consistent policies only terminate once the transactions are committed. Additionally, policies read-after-committed and speculatra 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. When compared to policy read-after-committed, policy speculatra, which uses Speculatra, consumed only 3%, 2% and 7% more memory than policy read-after-committed in in scenarios same-order, reorder-safe and reorder-conflict respectively.

Overall, the small overhead of Speculatra, given the additional guaranties it provides, is quite acceptable for most applications.

5.3.2 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 5.2. 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 Add, Subtract, Multiply, Divide, or Clear. 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 \((n_3 > 1)\). We did not explicitly analyzed the selective-writers variant because, for the same number of writers, its performance would be very similar to the multiple-writers variant considering they rely on the same base mechanism.

For this evaluation, we used the configuration described in Section 5.1, 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, due to high variance. The results of this evaluation can be seen in Figure 5.4, Figure 5.5 and Figure 5.6.
As expected, read times and resource consumption increase with the number of transactions per second (Figure 5.5. This is because monitoring the state of transactions is the most computation and memory-intensive task performed. However, the number of reads per second (Figure 5.6) and the number of writers (Figure 5.4) do not have a significant impact on the performance of our solution. We could potentially expect a difference between a single writer and multiple writers, as the *multiple-writers* variant checks all incoming pending transactions to determine if they invoke the contract, while the *selective-writers* variant does not require this. The overhead of this task is not affected by the number of writers. The reason we do not see any significant difference is that this task is relatively inexpensive. 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 [17].

Figure 5.4: Read time and resource consumption response to an increase in the number of simultaneous writers ($n_3$).

Figure 5.5: Read time and resource consumption response to an increase in the number of transactions per second ($n_1$).
5.3.3 Timeout policies

Monitoring the state of transactions is the most computationally expensive task performed by Speculatra. However, to minimize read latency, we must detect, as soon as possible, when a transaction has been mined. Considering this, we compared three different methods implemented for polling the state of pending transactions:

1. **Fixed timeout.** The polling timeout is previously defined and does not change over time. This is the method used in previous experiments.

2. **Dynamic timeout.** The polling timeout is dynamically adjusted over time according to the current mining rate. It is updated every time a new block is mined and set to half the average time between newBlockHeader events.

3. **New Block.** The blockchain state is polled when a new block is generated. With this approach, there is no timeout.

We ran these experiments using a workload that simulates a high load (4 transactions/s,
1 read/s, 4 writers) and a low load (0.5 transactions/s, 1 read/s, 4 writers), defined based on the results obtained in Section 5.3.2. For this experiments, we used 4 nodes connected to the Ethereum test network, Ropsten [18], which better approximates the main Ethereum network. At the time the experiments were conducted, the average mining rate in the test network was of 13 seconds. Therefore we ran experiments for fixed timeouts between 2 and 14 seconds, to determine which was the ideal value of this method.

Figure 5.7 shows that the latency is significantly smaller for a timeout interval of 2 seconds when compared to the other timeout values analyzed, for both high and low loads. Therefore we use a timeout of 2 seconds for the fixed timeout policy. The results presented in Figure 5.8 show that, for low loads, the method that simultaneously provides the lowest latency and overhead is a fixed timeout. This is because the fixed timeout was experimentally configured for optimal performance. The dynamic method eventually stabilizes with a similar timeout value. For high loads, Figure 5.8 shows that both the fixed and dynamic timeout methods present comparable results across all metrics, and perform better than the new blocks method.

![Figure 5.7: Latency comparison for fixed timeout policies.](image)

(a) Read Time (b) CPU Utilization (c) Memory Utilization

Figure 5.8: Read time and resource consumption of the three different timeout policies compared.
5.4 Speculative Transactions Overhead

To evaluate the overhead, in terms of gas consumption, of speculative transactions, we considered the traditional Smart Contracts defined in Listing 3.1 and Listing 5.1. For each Smart Contract, we considered two additional Smart Contracts, which support speculative transactions:

1. Speculatra wrapper Smart Contract of the original contract (Listing 4.4 and Listing 5.2, resp.)
2. Speculatra variant of the original Smart Contract (Listing 4.3 and Listing 5.3, resp.)

The first Smart Contract allows us to determine the overhead of supporting speculative transactions for existing Smart Contracts. The second Smart Contract allows us to determine the overhead of supporting speculative transactions for new Smart Contracts. We compared the cost of transactions invoking the corresponding methods in each of the Smart Contracts, in the Ethereum test network, Ropsten [18].

Table 5.1 shows the results of this evaluation. As expected, speculative transactions consume more gas than traditional transactions. This is because the gas used in a transaction depends on the number and the type of instructions executed during the transaction’s execution, and speculative transactions require additional instructions to validate the speculation.

Using the Speculatra wrapper Smart Contract, the speculative transactions consume 17% more gas than traditional transactions. With the Speculatra variant of the Smart Contract, speculative transactions consume only 6% more gas than traditional transactions. This difference is justified by the fact that calling other Smart Contracts 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, as additional gas consumed for the additional instructions corresponds to a smaller percentage of the transaction’s overall gas consumption.

5.5 Overall Performance

In this section, we focus on the evaluation of the overall execution of the DApp, considering both phases (see Figure 5.1), and determine the overall performance gain Speculatra can achieve.

For each experiment, we considered four different variants of the DApp described in Section 5.2:

- **Relaxed.** Follows read policy read-after-mined. This represents the more relaxed approach, which does not guarantee sequential consistency. This is the baseline for Speculat-
<table>
<thead>
<tr>
<th>Original</th>
<th>Wrapper</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>addToCart</td>
<td>22 527</td>
<td>26 157</td>
</tr>
<tr>
<td>checkoutCart</td>
<td>23 924</td>
<td>27 484</td>
</tr>
<tr>
<td>Add</td>
<td>42 338</td>
<td>46 141</td>
</tr>
<tr>
<td>Subtract</td>
<td>27 228</td>
<td>30 769</td>
</tr>
<tr>
<td>Multiply</td>
<td>24 584</td>
<td>30 816</td>
</tr>
<tr>
<td>Divide</td>
<td>24 589</td>
<td>28 044</td>
</tr>
<tr>
<td>Clear</td>
<td>23 545</td>
<td>29 659</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of gas cost of traditional transactions with the gas cost of speculative transactions

Figure 5.9: Execution Time for $P(\text{same-order}) = 90\%$, $P(\text{reorder-safe}) = 5\%$ and $P(\text{reorder-conflict}) = 5\%$.

- **No Speculation.** Follows read policy *read-after-committed*. This represents the traditional approach for ensuring sequential consistency.

- **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 5.3.1 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 5.3.1, we plotted the total execution time in terms of the processing time.

Figure 5.9, Figure 5.10 and Figure 5.11 show the results for $P(\text{same-order, reorder-safe, reorder-conflict}) = (90\%, 5\%, 5\%)$, $P(\text{same-order, reorder-safe, reorder-conflict}) = (90\%, 8\%, 2\%)$ and $P(\text{same-order, reorder-safe, reorder-conflict}) = (90\%, 2\%, 8\%)$, respectively. The results show that Speculatra can reduce the execution time up to 80% when compared to the traditional approach for ensuring sequential consistency. For very large processing times, close to the commit time, Speculatra \textit{Light} achieves similar performance to the full Speculatra solution. For shorter processing times, the full Speculatra solution can achieve a speedup of up to 80% when compared to Speculatra \textit{Light}.

5.6 Summary

The obtained results allows to conclude that Speculatra can effectively achieve similar performance to solutions that provide weaker consistency guarantees, with minimal overhead. Through speculation, Speculatra can reduce read latencies by more than 85%, compared to traditional solutions that also guarantee strong consistency.

We also evaluated the scalability of Speculatra for an increasing frequency of transactions and speculative reads, as well as concurrent clients issuing transactions. Results show that
Speculatra only the frequency of transactions has a minimal impact on the performance of Speculatra.

From the three different methods for polling the state of pending transactions, we determined that using a fixed polling interval of 2 seconds is the most efficient.

Speculative transactions, supported by Speculatra, consume 17% more gas for existing Smart Contracts and for new contracts, only 6% more.

Overall, by leveraging speculative transactions, Speculatra can provide up to 80% speedup on representative applications.
contract SpeculatraCalculator is speculatra {

    // Address of original contract
    address addr = 0x91a80aba0d4C2507B04B2736dbDE357451A8349D;

    function Add(uint _blockNumber, bytes32 _blockHash, int input)
        public verifySpeculation(_blockNumber, _blockHash)
    {
        Calculator calculator = Calculator(addr);
        calculator.Add(input);
    }

    function Subtract(uint _blockNumber, bytes32 _blockHash, int input)
        public verifySpeculation(_blockNumber, _blockHash)
    {
        Calculator calculator = Calculator(addr);
        calculator.Subtract(input);
    }

    function Multiply(uint _blockNumber, bytes32 _blockHash, int input)
        public verifySpeculation(_blockNumber, _blockHash)
    {
        Calculator calculator = Calculator(addr);
        calculator.Multiply(input);
    }

    function Divide(uint _blockNumber, bytes32 _blockHash, int input)
        public verifySpeculation(_blockNumber, _blockHash)
    {
        Calculator calculator = Calculator(addr);
        calculator.Divide(input);
    }

    function Clear(uint _blockNumber, bytes32 _blockHash)
        public verifySpeculation(_blockNumber, _blockHash)
    {
        Calculator calculator = Calculator(addr);
        calculator.Clear();
    }

    function Value() constant returns (int){
        Calculator calculator = Calculator(addr);
        return calculator.Value();
    }
}

contract ModifiedCalculator is speculatra {
    int public Value;

    constructor() public
    {
        Value = 0;
    }

    function Add(uint _blockNumber, bytes32 _blockHash, int input) public verifySpeculation(_blockNumber, _blockHash)
    {
        Value += input;
    }

    function Subtract(uint _blockNumber, bytes32 _blockHash, int input) public verifySpeculation(_blockNumber, _blockHash)
    {
        Value -= input;
    }

    function Multiply(uint _blockNumber, bytes32 _blockHash, int input) public verifySpeculation(_blockNumber, _blockHash)
    {
        Value *= input;
    }

    function Divide(uint _blockNumber, bytes32 _blockHash, int input) public verifySpeculation(_blockNumber, _blockHash)
    {
        if(input == 0)
            revert();
        Value /= input;
    }

    function Clear(uint _blockNumber, bytes32 _blockHash) public verifySpeculation(_blockNumber, _blockHash)
    {
        Value = 0;
    }
}

Listing 5.3: Speculatra variant Smart Contract for Solidity Smart Contract in Listing 5.1.
Chapter 6

Conclusions

Modern blockchain technologies, such as Ethereum, support the development of complex applications that take advantage of blockchain decentralized nature and tamper-proof resistance, through smart contracts. As a result, they show much potential to solve problems across many different business sectors, from finance to health-care management. Despite their increasing popularity, smart contract platforms still have many issues: from the lack of third-party tools and for supporting the development and testing of blockchain applications, limited community support and documentation, and performance limitations, among others.

A long-standing problem in blockchains is the high commit times. Due to this, and to ensure strong consistency, blockchain applications suffer from high latencies, which might be insupportable for some. As a result, some applications opt for relaxed approaches which, while present significantly better performance, can result in incorrect behavior.

Additionally, our review of the current programming model offered by Ethereum shows that programmers must understand the internal data structures and behaviors of the blockchain in order to develop blockchain applications. As a result, the application logic is mixed with the complex consistency management logic, leading to complex and error-prone code.

To address this problems, we introduced 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 smart contracts can be modified to support speculative transactions. This solution can also be extended to currently deployed smart contracts. Speculatra also effectively isolates the low-level details of blockchain state management from the application logic. This significantly reduces development effort and allows for simpler, more efficient and less error-prone code.
We evaluated the performance, overhead and scalability of Speculatra, and showed how applying speculation to determine the outcome of blockchain transactions through Speculatra can substantially decrease read latency with minimal overhead, while providing strong consistency. If the speculation is correct, Speculatra can reduce the read latency by more than 85%, when compared to the traditional approach for support sequential consistency. By capitalizing on speculative transactions, applications can experience an overall speedup of up to 80% on representative applications.

We showed that Speculatra scales well in terms of the number of transactions per second and that the number of current writers and the number of reads per second has a negligible impact on the performance and overhead of Speculatra.

Finally, our evaluation showed that Speculatra can support speculative transactions with only 6% gas consumption overhead.

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

6.1 Future Work

In this section, we discuss some possible future developments that could be made to improve our work.

- **Support more consistency models.** Speculatra could be expanded to support different consistency models, capitalizing on the variable consistency supported by blockchains.

- **Support Speculatra for other programming languages.** The implementation of Speculatra could be extended to other programming languages, with particular interest to languages directed at backend development such as Python or Java.

- **Improve detection of misspeculations.** Speculatra is able to reliably detect all misspeculations. However, in some scenarios, Speculatra might wrongly detect a mispeculation, which has not occurred (see Section 4.5). Additionally, the algorithm currently used by Speculatra to detect mispeculation could be further optimized by reducing the instances on which the call must be re-executed.

- **Improve the mechanism for identifying indirect updates to the smart contract.** Speculatra assumes that the developer is able to identify the smart contracts which invoke the application’s smart contract. However, during the application’s lifetime, this set of
contracts may change. Additionally, Speculatra currently does not account for the fact that some of the transactions of these smart contracts might not invoke the application’s smart contract at all. Ideally, this mechanism should be replaced by a method that allows Speculatra to efficiently identify smart contract internal transactions.

- **Restrict the set of tracked transactions.** Currently, Speculatra identifies all transactions which invoke a smart contract as potential updates to the contract state. However, not all transactions which invoke a smart contract modify its state. By refining the algorithm used to identify the transactions which update the smart contract state, we could potentially further reduce the read latencies achieved by Speculatra, decrease the memory overhead of Speculatra and the number of potential conflicts for each speculation. Some alternatives for this were discussed in Section 4.3.

- **More fine-grained speculation dependencies.** Currently, Speculatra considers the same set of tracked transactions when determining the dependencies of a speculation request. However, applications may wish to consider a different set of transactions for different requests. Speculatra’s implementation could be extended to support this.

- **Improve Speculative Transactions.** The current framework for developing smart contracts that support speculative transactions could be optimized to ensure the most efficient gas consumption possible.

- **Extend evaluation of Speculatra.** Speculatra could be evaluated for a broader and more diverse set of DApps.

- **Support Speculatra for other blockchains based on EVM.** Speculatra could be implemented for other blockchain technologies which are also based on the Ethereum Virtual Machine (EVM), e.g. Quorum [76].
Bibliography


