An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts

Vikram Saraph
Department of Computer Science, Brown University, USA
vsaraph@cs.brown.edu

Maurice Herlihy
Department of Computer Science, Brown University, USA
mph@cs.brown.edu

Abstract

We use historical data to estimate the potential benefit of speculative techniques for executing Ethereum smart contracts in parallel. We replay transaction traces of sampled blocks from the Ethereum blockchain over time, using a simple speculative execution engine. In this engine, miners attempt to execute all transactions in a block in parallel, rolling back those that cause data conflicts. Aborted transactions are then executed sequentially. Validators execute the same schedule as miners.

We find that our speculative technique yields estimated speed-ups starting at about 8-fold in 2016, declining to about 2-fold at the end of 2017, where speed-up is measured using either gas costs or instruction counts. We also observe that a small set of contracts are responsible for many data conflicts resulting from speculative concurrent execution.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies

Keywords and phrases Blockchains, Smart Contracts

Digital Object Identifier 10.4230/OASIcs.Tokenomics.2019.4

Related Version `https://arxiv.org/abs/1901.01376`

1 Introduction

A *blockchain* is a distributed data structure that implements a *ledger*: a tamper-proof, widely-accessible, append-only sequence of *transactions*. Blockchains form the basis for cryptocurrencies [8, 10, 11, 12, 13] and other applications that must maintain a shared state in the absence of a trusted central authority. In the Ethereum [8] blockchain, for example, if Alice wants to send a coin to Bob, she broadcasts her transaction to one or more *miners*, who package transactions into *blocks*, and then undertake a consensus protocol to agree on which block should be appended next to the shared blockchain. A *validator* is any party who reads the blockchain state and checks it for correctness. Miners are validators, of course, but so is any party who needs to query the blockchain state.

In many blockchain systems, client transactions can invoke scripts, often called *smart contracts*, or just *contracts*, that perform logic needed to support complex services such as trading, voting, and managing tokens. Here, we focus on Ethereum-style smart contracts.

Ethereum's smart contracts present a concurrency challenge. To reconstruct the blockchain's current state, each validator must re-execute, in a sequential, one-at-a-time order, every call to every smart contract. Such sequential validation is unattractive because it fails to exploit the concurrency provided by modern multicore architectures. Simply executing those calls in parallel is unsafe, because there may be dependencies between contracts: if one contract depends on the results of another, then those contracts must be executed in the same order by every validator. Because the Ethereum smart contract language is Turing-complete, and because contracts can reference one another through untyped function pointers, static analysis is unlikely to be broadly effective.

International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019).
Editors: Vincent Danos, Maurice Herlihy, Maria Potop-Butucaru, Julien Prat, and Sara Tucci-Piergiovanni;
Article No. 4; pp. 4:1–4:15
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The most promising approach to concurrent execution of smart contracts is *speculation* [1, 4, 7]: the virtual machine executes contract calls in parallel against the current state, tracking each transaction's *read set* and *write set* (memory locations read and written). Writes to memory are intercepted and buffered. Two transactions *conflict* if they access the same memory location, and one access is a write. For every pair of conflicting transactions, one is discarded, and the other is committed. Speculative techniques typically work well when conflicts are rare, but perform poorly when conflicts are common.

1.1 Contributions

How well does speculative concurrency work for smart contract execution? This paper makes the following contributions. We exploit publicly-available historical data to estimate conflict rates in an existing blockchain. This methodology, replaying the historical transaction record against proposed alternative run-times, could be a useful model for other blockchain-centered investigations. This study is *exploratory*: it aspires to provide a relatively fast and cheap estimate of how well certain strategies are likely to do in practice, with the goal of focusing future research attention in directions more likely to be productive. Of course, an exploratory study necessarily employs sampling, estimation, and approximation.

As described below in more detail, we re-execute blocks sampled from the Ethereum blockchain against a simple speculative execution engine. This engine has two phases: in the first (concurrent) phase, all transactions are run in parallel. In the second (sequential) phase, transactions observed to conflict are discarded and re-run in one-at-a-time order. This execution strategy produced speed-ups ranging from about 8-fold for blocks sampled from 2016, gradually declining to about 2-fold for blocks sampled from 2017.

This study makes the following observations.

- Even simple speculative strategies yield non-trivial speed-ups.
- Over time, however, these speed-ups declined as transaction traffic increased.
- Distinguishing between reads and writes is important: treating a transaction's read and write sets as a single conflict set substantially increases conflict rates.
- More aggressive speculative strategies, such as running multiple concurrent phases, yield little additional benefit.
- Accurate static conflict analysis may yield a modest benefit.
- Increasing the number of cores in the simulated virtual machine from 16 to 64 improved speed-ups, but there was little improvement above 64 cores.
- In high-contention periods, most contention resulted from a very small number of popular contracts.

These observations suggest some directions for further research.

- In periods of high contention, most conflict is caused by a small number of very popular contracts. Today, contract writers have no motivation for avoiding such conflicts. It could be productive to devise incentives, perhaps in the form of reduced gas prices, for contracts that produce fewer data conflicts.
- Many data conflicts, such as crediting and debiting account balances, are probably artifacts of defining conflict naïvely in terms of read-write sets. Perhaps conflicts could be reduced by extending the virtual machine to provide explicit support for common commutative operations such as credits and debits.

1.2 Methodology in Brief

We replay each transaction in a block, computing each transaction's read and write sets. We then greedily sort the transactions into two bins: the *concurrent bin* holds transactions that do not conflict with any other transaction already in the concurrent bin, and the *sequential bin* holds the rest.

We then estimate the elapsed time required to (1) execute the concurrent bin transactions in parallel (including the cost of detecting and discarding conflicting transactions), followed by (2) sequentially executing the sequential bin transactions.

Since we do not have a parallel EVM implementation to test, we estimate a transaction's running time in two ways: either by the *gas* it consumed, or by the number of Ethereum Virtual Machine (EVM) bytecode instructions executed. (Both measures are easy to compute, and yield similar results.) The speed-up is the ratio between the estimated elapsed times for the sequential executions versus the longest speculative execution.

In the next section we describe related work. In Section 3, we outline Ethereum's architecture, smart contracts, and all relevant terminology. Section 4 describes the setup of our empirical study along with statistics summarizing observed results, while in Section 5, we consider various alternatives to the baseline setup. Finally, in Sections 6 and 7, we discuss conclusions and potential future directions for extending this work.

2 Related Work

Smart contracts were first proposed by Szabo [15].

Bitcoin [12] includes a scripting language of limited power. Ethereum [8] is perhaps the most widely used smart contract platform, running on a quasi-Turing-complete virtual machine. Solidity [14] is the most popular programming language for the Ethereum virtual machine. Other blockchains that support smart contracts include Corda [5] and Cardano [9].

Hyperledger Fabric [4] is a permissioned blockchain where transactions (calls to smart contracts) are executed speculatively in parallel against the latest committed state. Transactions' read and write sets are recorded and compared, and conflicting contracts are discarded.

Dickerson et al. [7] have proposed a speculative execution model where miners dynamically construct a fork-join schedule that allows concurrent executions without violating transaction dependencies. Anjana *et al.* [1] propose a way to extend this approach to lock-free executions.

3 Ethereum Smart Contracts

In this section, we provide some background on Ethereum smart contracts, mining, and validation of Ethereum blocks.

3.1 The Architecture

In Ethereum, as in other blockchains, multiple *nodes* follow a common protocol in which *transactions* from *clients* are packaged into *blocks*, and nodes use a consensus protocol to agree on successive blocks. Each block includes a cryptographic hash of its predecessor, making it difficult to tamper with the ledger.

Each client has ownership of one or more *accounts*, so that each transaction occurs between a *sender* account and a *recipient* account. The majority of transactions are one of two kinds: either a *value transfer*, which is a purely monetary transfer of *ether* from sender to recipient, or a *contract call*, where the sender account makes a call to code associated with the recipient account.

```
contract StorageInterface {
    mapping (uint => uint) storage;

    function getValue(uint key) returns (uint) {
        return storage[key];
    }

    function setValue(uint key, uint value) {
        storage[key] = value;
    }
}
```

(a) This code is a simple smart contract written in the Solidity language. The contract, StorageInterface, contains a Solidity mapping named storage, and has functions getValue and setValue. Mappings are essentially hash tables that store key-value pairs, and are the primary means of accessing Ethereum contract storage. Figure 1b shows a snippet of the bytecode after compiling the Solidity contract. The bytecode is what is published in the blockchain.

```
 ...
JUMPDEST
DUP1
PUSH1 0x0
DUP1
DUP5
DUP2
MSTORE
PUSH1 0x20
ADD
SWAP1
DUP2
MSTORE
PUSH1 0x20
ADD
PUSH1 0x0
KECCAK256
DUP2
SWAP1
SSTORE
POP
 ...
```

(b) Bytecode resulting from compiling the contract.

Figure 1 An example Solidity smart contract and a fragment of the corresponding bytecode.

Some Ethereum accounts, in addition to maintaining a balance of ether, possess associated code called a *smart contract*. A smart contract resembles an object in a programming language, with a long-lived *state* recorded in the blockchain. This state is manipulated by a set of *functions* called either directly by clients (top-level calls) or indirectly by other smart contracts (internal calls). To ensure that function calls terminate, each computational step incurs a cost in *gas*, paid by the caller. The caller specifies a maximum amount of gas it is willing to pay, and if the charge exceeds that sum, the computation is terminated and rolled back, and the caller's gas is not refunded. Nevertheless, the rolled-back transaction is still recorded on the blockchain.

A smart contract's code consists of a sequence of bytecode instructions, taken from the Ethereum bytecode instruction set. Every bytecode instruction consumes a certain amount of gas. Each client runs an instance of the Ethereum virtual machine (EVM), which executes calls to smart contracts and runs their sequence of instructions. While users may author smart contracts in higher level languages such as Solidity, these contracts must ultimately be compiled into EVM bytecode, since it is the bytecode that is published in the blockchain. The virtual machine specification, the bytecode instruction set, and all associated gas costs are described in Ethereum's "Yellow Paper" [16].

3.2 Mining and Validation

Smart contracts are first executed by *miners*, or nodes that repeatedly propose new blocks to append to the blockchain. When a miner creates a block, it selects a sequence of client transactions and executes their smart contract codes in sequence, transforming the old contract state into a new state.

Once a block has been appended to the blockchain, that block's smart contracts are re-executed by *validators*, or nodes that reconstruct (and check) the current blockchain state. Each miner validates blocks proposed by other miners, and older blocks are validated by newly-joined miners, or by clients querying the contract state. Once a contract is in a block, it is effectively re-executed forever (in Ethereum), so contract executions by validators vastly exceed executions by miners.

As noted earlier, one drawback of the Ethereum protocol is that a block's contracts are executed in a one-at-a-time order, so miners and validators cannot exploit modern multicore architectures. Contracts cannot be executed concurrently in a naïve way, because they share storage, and may be subject to *data conflicts*, that is, concurrent accesses to the same the storage variables.

In the absence of explicit concurrency guidelines, we execute transactions *speculatively* in parallel, allowing non-conflicting contracts to commit, but rolling back conflicting transactions, and running them sequentially in a second phase.

A novel aspect of this study is that we analyze the effectiveness of speculation against the *historical* record of transactions actually executed on the Ethereum blockchain, replaying their bytecode instructions. We are not aware of an Ethereum virtual machine implementation that supports concurrency, so we simulate concurrent transaction execution by stepping through each transaction's instructions, using eager conflict detection to sort each block's transactions into a conflict-free parallel bin, and a conflicted sequential bin. This strategy is simple and scalable, a natural starting point for an empirical study.

4 The Baseline Experiment

Here we describe the baseline experiment testing the effectiveness of a simple speculative execution strategy against historical transaction data from the Ethereum blockchain. In Section 5, we describe variations on this baseline strategy that probe the sensitivity of our measurements, as well as alternative speculative strategies.

4.1 Setting up the Experiment

We set up an Ethereum node by installing and running the Ethereum Go client, also known as `geth`. The `geth` client allows one to synchronize with other network nodes and reconstruct the blockchain by validating each block. The client comes packaged with an interface for fetching data from the blockchain, as well as debugging tools for inspecting transactions. The debugging API includes a utility for reproducing the bytecode trace of a given transaction.

The `geth` client can run in various synchronization modes, which determine what and how much old state the client records. For example, a client running in *light* mode will keep track of the blockchain's current state only, while in *archive* mode, the client maintains the full log of all previous states.

Using the `geth` API, it is straightforward to retrieve any given smart contract's bytecode. However, the only way to tell how that bytecode affects the blockchain state is by re-executing each transaction before it. Therefore we use `geth`'s tracing utility to re-execute

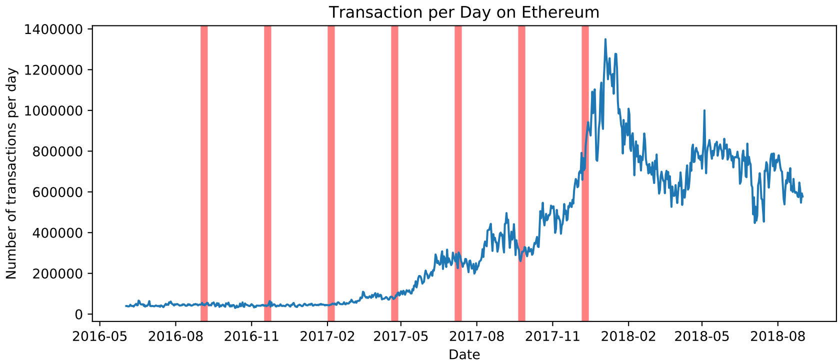


Figure 2 Historical periods analyzed superimposed on number of transactions per day.

Period	Interval	Blocks	Transactions	Calls
1	Sept 1 – 8, 2016	4200	32502	11779
2	Nov 17 – 24, 2016	4200	32648	11070
3	Feb 2 – 9, 2017	3600	26415	10586
4	Apr 20 – 27, 2017	4100	56051	23684
5	July 6 – 13, 2017	3400	197003	84219
6	Sept 21 – 28, 2017	2100	200980	96043
7	Dec 7 – 14, 2017	4100	557471	255872

Figure 3 Number of blocks, transactions, and contract calls analyzed in each historical period.

transactions, running the client in archive mode so it can reference past states. We found that reconstructing the blockchain from scratch in this way was much too slow to be feasible on an ordinary computer (an archive blockchain sync could not even keep up with the current blockchain growth rate), so we obtained a baseline copy of the blockchain from the ConSenSys archive [3] dating from Ethereum's origins in 2016 to early October 2017. Starting from this base, we used the `geth` utilities to synchronize up to December 2017.

Between these dates, the Ethereum blockchain contains about 4 million blocks and 100 million transactions, far too much data to analyze in detail in a reasonable time with reasonable resources. Instead, we chose to focus on seven historical periods between July 2016 (after the DAO fork) and December 2017. Each period spans roughly one week, with consecutive periods separated by 11 weeks, so that each day of the month is considered. Due to computational limitations, we analyze every 10th block in each of the seven historical periods. Figure 2 shows these periods superimposed on the number of transactions per day at that time.

4.2 The Greedy Concurrent EVM

We simulated a concurrent EVM that executes transactions speculatively in parallel using the following *greedy* strategy. For each block, execution proceeds in two phases, an initial *concurrent phase*, and a subsequent *sequential phase*. We consider execution engines with either 16, 32, or 64 threads. In the concurrent phase, each thread chooses a transaction from the block and executes it speculatively. If that transaction encounters a conflict, the transaction's effects are rolled back, and that transaction is deferred to the second, sequential phase. When a thread finishes executing a transaction, it picks another to execute, continuing until all transactions in the current block have been chosen.

The second phase starts when the first phase is complete: the transactions that encountered data conflicts in the first phase are re-executed sequentially. In the second phase, data conflicts are not an issue since transactions are explicitly serialized and state changes committed sequentially.

Two transactions *conflict* if they access the same storage location, and at least one access is a write. Transactions that do not conflict are said to *commute*, because interleaving them in any order yields the same transaction and storage states. At a more granular level, two bytecode operations *conflict* if applying them in different orders yields different storage states. Most bytecode operations, such as arithmetic operations and others that interact with local state, commute with one another. Bytecode operations that interact with shared state, on the other hand, can potentially conflict.

The EVM operations SLOAD and SSTORE read from and write to persistent storage, respectively, and are used for nearly every state access and state modification. By far the most common conflicts arise from conflicting these two operations. Other kinds of conflicts, while possible, are assumed to be too rare to monitor. For example, one transaction might create a new contract, while another calls the newly created contract in a way that creates a race condition: the call may or may not arrive before the contract is initialized. There is also a bytecode operation that reads a given account's balance, which is part of the blockchain's shared state.

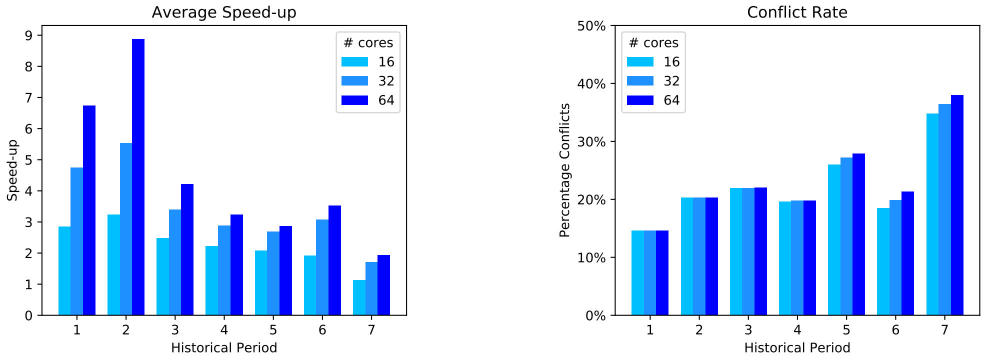
We detect conflicts by associating a *read-write lock* with each storage location. Each SLOAD (respectively, SSTORE) operation requests that location's lock in read (write) mode. If a transaction requests a lock that is already held in a conflicting mode by another transaction, the requesting transaction is rolled back and deferred to the next phase. No locks are released until the concurrent phase ends, even those held by aborted transactions. This is to ensure that no interleaving of transactions in the concurrent bin result in a conflict when re-executed by validators.

4.3 Sampling and Evaluation

In this section, we evaluate the concurrent speculative execution strategy on the data set of transaction traces as described in the experimental setup (see Section 4.1). Traces from the historical Ethereum blockchain are used to simulate what would have happened if the original blocks of transactions were instead re-executed concurrently.

Our principal figure of merit is *speed-up*: the ratio between the time to execute a block's transactions sequentially, and the time to execute the same transactions concurrently and speculatively. For now, we use cumulative gas costs as a proxy for time. Then to estimate speed-up, we measure the ratio between (1) the gas consumed to execute a block's transactions sequentially, versus (2) the maximum gas used by any thread in the parallel phase plus the gas needed to execute the sequential phase. Note that an aborted transaction may be counted twice: once (partially) in the concurrent phase, up to when the transaction is aborted, and once in the sequential phase when the aborted transaction is re-executed in isolation.

4.4 Baseline Results



(a) Average speed-up.

(b) Average conflict rate.

Figure 4 Average speed-up and conflict rate for each of the seven historical time intervals. Increased Ethereum activity over time is reflected by decreasing speed-up and higher conflict rates.

Figure 4 summarizes speed-up statistics over the seven historical periods, along with the *conflict rates*, or the percentage of contract calls that abort per block. The average speed-up and conflict rate are shown for simulated VMs of 16, 32, and 64 cores, where averages are weighted by the number of contract call transactions in each block.

Earlier periods display higher speed-ups and lower conflict rates because transaction volume and contention are low. For example, speed-up is as high as 3.23 in the second historical period on a simulated EVM with 16 cores, and this number rises to 8.87 for a 64 core EVM. During the same interval, contract calls abort at a rate of only 20%.

As the volume of transactions increases over time, however, so does the rate of transaction conflict, so more time is spent sequentially re-executing transactions aborted during the concurrent phase. This naturally leads to lower speed-ups, since there is less opportunity to parallelize transaction execution. Indeed, during the December 2017 period, with 16 threads, roughly 34% of transactions abort. Nevertheless, it is notable that there is still a modest but positive speed-up of 1.13 even then. Moreover, this speed-up effectively doubles, to slightly more than 2 when there are 64 threads. When transaction volume is higher, using more threads yields more speed-up.

4.5 Speed-up Distributions

The average speed-up of each historical period provides little insight into how the blocks' speed-ups are distributed in each period. Here, we further analyze the performance of speculative execution by looking at the distributions of these speed-ups. Due to space limitations, we focus on historical periods 5, 6, and 7, since these have the highest transaction volumes.

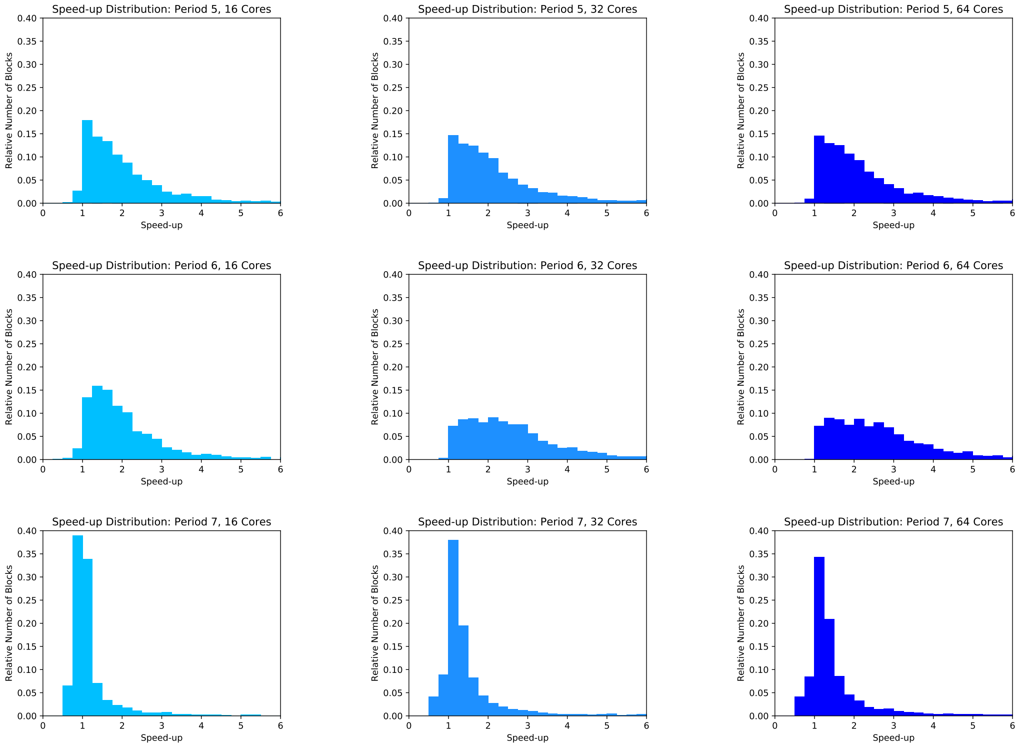


Figure 5 Speed-up distributions for each combination of historical periods 5, 6, and 7, and with 16, 32, and 64 cores. Each plot depicts one such combination. Blocks are binned in increments of 0.25 according to speed-up realized. Distributions are normalized so that total area is roughly 1.

Speed-up distributions of the three periods are depicted in Figure 5. Each plot is normalized to represent an approximate probability density, so that the y-axis is interpreted as a relative count. Since there is relatively less contention in Periods 5 and 6, their speed-up distributions are flatter, with longer tails. This is because more blocks during these periods are able to exhibit a higher amount of speed-up. However, when the transaction rate is higher, as in seventh period of December 2017, the distributions are concentrated closed to 1, since more work is wasted as the number of aborted transactions rises.

As noted before, some blocks may realize a slowdown when using our concurrent execution strategy, a result of the cost of aborting and re-executing conflicting transactions. The relative number of such blocks is shown in plots of Figure 5, where regions to the left of 1 represent blocks that are slowed down. Making more cores available to the EVM appears to reduce the number of slowed-down blocks for each of the three historical periods. Though fewer than 3% blocks are slowed down for both Periods 5 and 6 when using 16 cores, this percentage jumps to 46% for Period 7. However, when the EVM's number of cores is increased to 32, the percentage of blocks that are slowed down drops substantially to only 13%.

4.6 Storage Hot-Spots

Next, we investigate transaction contention by analyzing how often certain storage addresses are accessed by transactions. In particular, we look at memory addresses that are conflict points for pairs of transactions, and how many times each such address results in a conflict. Addresses that attract a high number of conflicts are informally called *hot-spots*.

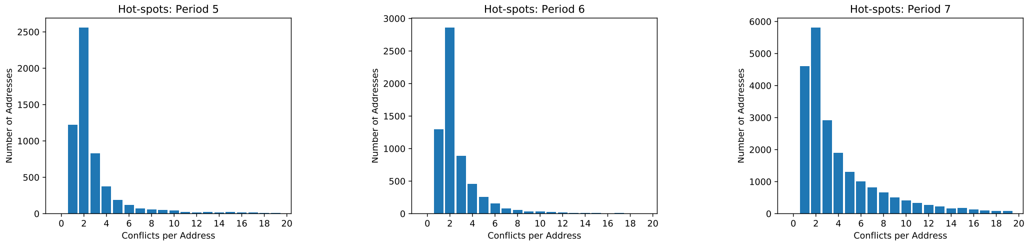


Figure 6 Histograms of conflicts per address. Addresses that results in a large number of conflicts are hot-spots. Period 7 has a much heavier tail, corresponding to many more address hot-spots.

In Figure 6, we plot histograms illustrating the number of conflicts per address, for historical periods 5, 6, and 7. Each storage address with at least one conflict is binned according to how many conflicts occurred at that address. For example, in Period 5, there are 2562 unique storage addresses at which there were exactly 2 conflicts. Periods 5 and 6 have relatively few storage addresses with high contention, and this pattern is similar to the earlier historical periods. However, Period 7's histogram has a much heavier tail than the other two histograms, meaning that there are many more storage addresses with larger numbers of conflicts. In other words, there is a handful of addresses that many different transactions attempt to access, most likely a result of so many transactions calling the same few contracts. We elaborate on this observations in Section 5.6 below.

5 Alternative Experiments

We extended the baseline experiment with a number of other experiments intended to test the effectiveness of alternative strategies, and to test the sensitivity of our approximations.

5.1 Sampling

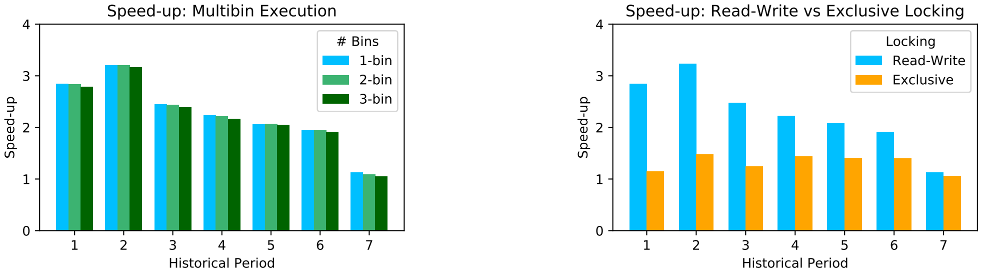
To test whether sampling 1 in 10 blocks during a period yielded a distorted view, we ran a detailed simulation of one period (Period 5) to compare the results with the sampled simulation. There is a modest difference in speed-up, and an even more negligible difference in abort rate. Some discrepancy can also be explained by the inherent randomness of a concurrent scheduler.

Sampling rate	**Speed-up**			**Abort rate**		
	16 cores	**32 cores**	**64 cores**	**16 cores**	**32 cores**	**64 cores**
1-in-10 sampling	2.063	2.694	2.859	25.98%	27.23%	27.88%
All blocks	2.085	2.717	2.913	25.96%	27.21%	27.86%

Figure 7 Accuracy of 1-in-10 sampling for historical period 5.

5.2 Multiple Phases

The greedy, two-phase strategy can be generalized to encompass multiple concurrent phases, so that each transaction that was deferred in one concurrent phase is instead re-executed in another subsequent concurrent phase. It is possible that multiple concurrent phases could provide additional speed-up. However, as illustrated in Figure 8a, we found in our experiments that executing two concurrent phases almost always yields less speed-up than executing a single concurrent phase. This decrease is due to the duplicate work performed by transactions rolled back in multiple phases, with not enough additional speed-up yielded in the latter concurrent phases. Therefore, in practice, one current phase is sufficient to realize almost all potential concurrency.



(a) Speed-up with multiple concurrent phases for transaction execution.

(b) Speed-up when the EVM uses mutex locks to lock storage addresses.

Figure 8 Multiple phases and data conflicts figures.

5.3 Data Conflicts

In our simulated concurrent EVM, transactions access storage addresses by first acquiring a read-write lock. This allows multiple transactions to read an address, with no conflict, if there are no concurrent writers. In principle, this decreases the number of conflicts when compared to using mutex locks, hence increasing the potential speed-up.

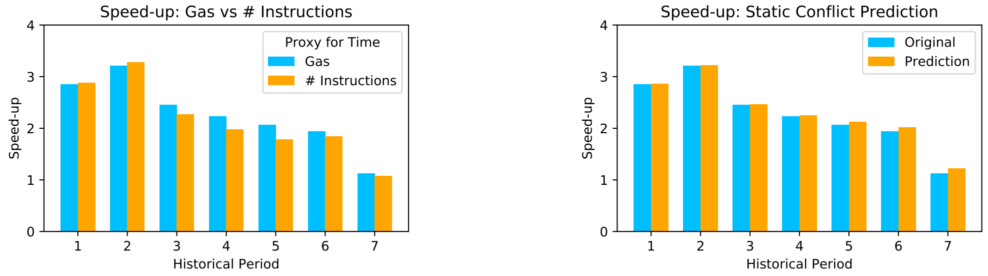
To determine whether read-write locks reduce data conflicts in practice, we investigated the effect of simplifying the conflict model by merging all data accesses into a single conflict set, by using mutex locks for conflict detection. With the exception of the last historical period in which speed-up is already low, the speed-ups were substantially worse than the speed-ups obtained by distinguishing between read and write accesses. See Figure 8b for a comparison of the two locking schemes when using 16 cores. These results suggest that there is significant value in implementing a concurrent EVM with read-write locks, instead of the simpler mutex locks.

5.4 Proxies for Time

Since we do not have access to an actual concurrent EVM, we must estimate how long it takes to execute each transaction. There are two straightforward choices: we can count the number of instructions executed by each transaction, or we can tally the gas cost of executing the transaction's instructions. The first choice assumes that each EVM instruction takes roughly the same time to execute, while the second assumes that instruction gas cost is roughly proportional to execution time. For example, the arithmetic operations MUL and DIV require 5 units of gas, while SSTORE and SLOAD cost 20000[1] and 200 units respectively.

All speed-ups reported so far were measured in terms of gas costs. As a sanity check, for 16 cores, we recomputed speed-ups using instruction count as a proxy for time. These speed-ups are shown in Figure 9a. There does not appear to be any significant qualitative difference between gas cost and instruction count.

[1] SSTORE costs 20000 units when storing a non-zero value, but costs only 5000 units otherwise.



(a) Speed-up when instruction count is used to measure transaction execution time.

(b) Speed-up under static conflict prediction. Aborted transactions costs effectively ignored.

Figure 9 Proxies for time and static conflict prediction figures.

5.5 Static Conflict Prediction

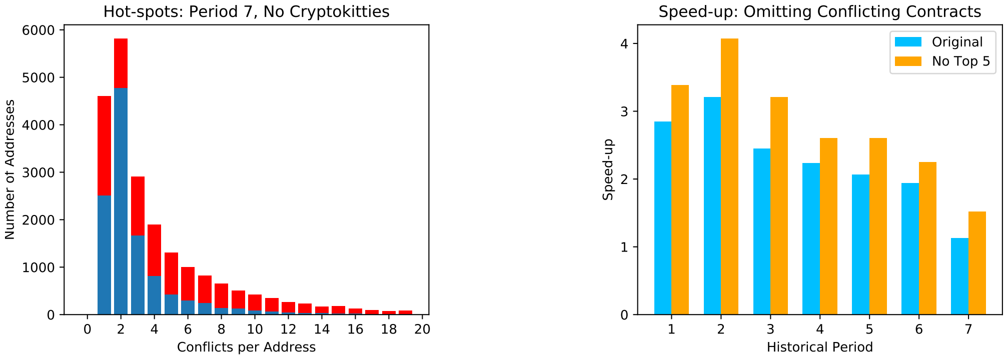
If we were able to predict whether a transaction would abort if executed speculatively, then we could save the cost of transaction roll-back and retry. We can simulate the effect of a perfect conflict predictor simply by ignoring the cost of aborted transactions, as in Figure 9b. However, doing so yields a negligible change in speed-ups, less than 0.1% in most cases. The only exception is the very last period (December 2017), where contention was very high. For that historical period, the average speed-up increases from 1.13 to 1.22. These numbers suggest that static conflict analysis, if accurate enough, may yield modest gains during periods of high contention.

5.6 Omitting Hot-spot Contracts

In most of the previous analysis, Period 7 stands out among the selected historical periods as being particularly high volume and contentious. Recall that this period was sampled from December 2017, which is close to peak Ethereum transaction activity. More specifically, Period 7 occurred when there was great interest from the general public over *CryptoKitties* [2] [6], a recreational game deployed on Ethereum in which users create, breed, and trade virtual cats. The popularity of CryptoKitties is especially apparent when using any blockchain explorer to browse Ethereum transactions during December 2017. The popularity of CryptoKitties was responsible for congesting the Ethereum network, though interest in it has since died down.

In light of the CryptoKitties frenzy, we reanalyzed Period 7 under a hypothetical scenario in which the CryptoKitties contracts [2] did not exist. This is easily accomplished by ignoring all calls to the CryptoKitties contracts when tracing each block's transactions and replaying them, and calculating the resulting statistics.

[2] The vast majority of calls to CryptoKitties are to its core contract (which has address hash `0x06012c8cf97bead5deae237070f9587f8e7a266d`), and to an auction contract (which has address hash `0xb1690c08e213a35ed9bab7b318de14420fb57d8c`).



(a) The effect on hot-spot counts when CryptoKitties contract calls are ignored.

(b) Speed-up when calls to the five most conflicting contracts are ignored.

Figure 10 Speed-up analysis after removing high-contention or high-volume contracts.

After filtering out all calls to CryptoKitties contracts, speed-up using 16 cores rises from 1.13 to 1.65. These calls accounted for about 31% of all contract calls, and furthermore, by filtering out CryptoKitties, the number of contract calls per block drops from 62 to 43, which is much closer to the 46 calls per block in Period 6. While a speed-up of 1.65 does not quite match speed-up from older periods, it is still clear from this simple analysis that much of the contention in Period 7 is caused solely by CryptoKitties.

To further illustrate the impact that CryptoKitties had, we reproduce the conflict histogram from the previous section, using the same hypothetical scenario with no CryptoKitties. Recall that the conflict histogram of Period 7 had a much heavier tail than the corresponding histograms of older intervals. However when CryptoKitties is removed, Period 7's histogram (Figure 10a) has a much thinner tail, resembling the other histograms. Indeed, in Period 6, 10% of conflicting storage addresses have at least 5 conflicts, but in Period 7, this same number is 30%. However, if CryptoKitties contracts are ignored, then only 14% or conflicting address have at least 5 conflicts. These numbers demonstrate that CryptoKitties is responsible for many storage hot-spots.

We generalize this analysis to the other historical periods by determining the top five most conflicting smart contracts in each period, and reproducing scenarios where none of the highly conflicting contracts were called. As shown in Figure 10b, this results in a noticeable speed-up in each period, not just the 7th. Most of these highly conflicting contracts are actually token contracts; in fact, the most contentious contract from each period is either a token (or a token exchange) contract. Therefore, analyzing these small sets of contracts may provide insight into how to reduce contention when speculatively executing smart contracts in parallel.

6 Discussion

As noted, this study is exploratory, incorporating various approximations and omissions. Most such omissions are the result of the absence of a standard concurrent EVM implementation. This study does not account for some EVM overheads, such as the costs for *value transfers*, where one account transfers ether directly to another, without modifying storage. Value transfers typically commute, and are likely much faster to execute than contract calls, though they do make up the majority of all Ethereum transactions. Other sources of overhead include solving a cryptographic puzzle to compute proof of work, which would affect miners' speed-up but not the validators'.

In the absence of a timing model for a concurrent EVM, this study uses gas costs (or instruction counts) as proxies for time when computing speed-up. As noted, both proxies yield essentially equivalent results.

This study relies on sampled blocks because the volume of data in the Ethereum blockchain is simply too large to make exhaustive analysis practical or rewarding. An archive synchronization of the blockchain was a major computational overhead for this study, in addition to recovering transaction traces. These overheads may be reduced as further Ethereum tools and utilities are developed.

There are several reasons why speculative parallelism may sometimes yield little, or even negative speed-up. For example, a block might contain one transaction substantially longer than the others, whose execution time dominates the block execution time. In this case, it is impossible to achieve much speed-up, no matter how these transactions are scheduled and distributed among multiple cores. Or if a block contains very few contract call transactions, there is little opportunity for speed-up. If a block's transactions all access the same storage location, perhaps because they all access the same popular contract [6], then speculation will produce a negative speed-up as a result of the cost of rolling back so many misspeculated transactions.

7 Conclusions

Our results suggest that a simple speculative strategy based on read-write set overlap can produce non-trivial speed-ups, but that such speed-ups will decline as transaction rates and conflict rates increase. More aggressive strategies, such as adding additional parallel phases, seem to provide little additional benefit, because conflict appears to be bursty: if one transaction conflicts with another, then it probably conflicts with multiple others.

The results of this study suggest that the most promising way to further increase parallelism in Ethereum-style smart contract execution is to reduce the conflict rate, perhaps by focusing on reducing unnecessary conflicts. We observed that splitting transactions' data sets into read sets and write sets decreased conflict rates substantially, suggesting that conflict rates are sensitive to the *semantics* of concurrent operations on shared data. This observation suggests that conflict rates might be reduced even further if the execution engine could do a better job of recognizing when operations commute at the semantic level. For example, transactions that increment or decrement the same account balance (a common occurrence) have overlapping read and write sets, and are therefore deemed to conflict. At the semantic level, however, these operations commute (in the absence of overflow or underflow), so as long as the virtual machine's memory operations are atomic, those operations need not conflict. (Our study could not detect which conflicts are real, and which are artifacts, because only compiled bytecode was available for analysis).

It might be profitable to investigate the effects of endowing the virtual machine with intrinsic data types such as atomic counters or atomic sets that provide many commuting mutator operations. Studying highly conflicting token contracts may provide insight into which kinds of data types or operations would best alleviate contention.

Periods of high contention and low speed-up are caused by a relatively small number of popular contracts. Currently, smart contract designers have no guidance how to avoid speculative data conflicts, nor any incentive to do so. Our results suggest that there is a need to devise incentives for smart contract programmers to design contracts in ways that reduce conflicts, either by eliminating spurious conflicts, or by exploiting improved commuting bytecode instructions.

References

1 Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An Efficient Framework for Concurrent Execution of Smart Contracts. *CoRR*, abs/1809.01326, 2018. arXiv:1809.01326.

2 Nellie Bowles. CryptoKitties, Explained... Mostly. *The New York Times*, December 2017. URL: https://www.nytimes.com/2017/12/28/style/cryptokitties-want-a-blockchain-snuggle.html.

3 Consensys. Harness the power of Ethereum. https://consensys.net/. As of 22 November 2018.

4 Hyperledger Consortium. HyperLedger Fabric Release 1.3. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.3/.

5 R3 Corda. Writing a contract. URL: https://docs.corda.net/releases/release-M1.0/tutorial-contract.html. As of 31 December 2018.

6 Crytokitties. CryptoKitties | Collect and breed digital cats! https://www.cryptokitties.co/. As of 22 November 2018.

7 Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding Concurrency to Smart Contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 303–312, New York, NY, USA, 2017. ACM. doi:10.1145/3087801.3087835.

8 Ethereum. https://github.com/ethereum/.

9 IOHK. Cardano Settlement Layer Documentation. https://cardanodocs.com/technical/plutus/introduction/. As of 31 December 2018.

10 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol, 2017. As of 24 November 2018 URL: https://iohk.io/research/papers/#ouroboros-a-provably-secure-proof-of-stake-blockchain-protocol.

11 David Mazieres. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus, 2015. As of 24 November 2018. URL: https://www.stellar.org/papers/stellar-consensus-protocol.pdf.

12 Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, May 2009. URL: http://www.bitcoin.org/bitcoin.pdf.

13 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm. https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

14 Solidity documentation. http://solidity.readthedocs.io/en/latest/index.html.

15 Nick Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), 1997. URL: http://firstmonday.org/ojs/index.php/fm/article/view/548.

16 Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.