Efficient Scheduling of Smart Contract Transactions via Conflict Graph Coloring

Ankit Ravish
Indian Institute of Technology
Hyderabad, Telangana, India
0009-0009-7355-8455

Yaron Hay
Technion – Israel Institute of Technology
Haifa, Israel
0009-0006-1263-7318

Piduguralla Manaswini
Indian Institute of Technology
Hyderabad, Telangana, India
0000-0002-6295-0445

Rishabh Jain
Indian Institute of Technology
Hyderabad, Telangana, India
0009-0002-1812-803X

Roy Friedman
Technion – Israel Institute of Technology
Haifa, Israel
0000-0001-6460-9665

Sathya Peri
Indian Institute of Technology
Hyderabad, Telangana, India
0000-0002-3471-7929

***Abstract*—A smart contract is a special type of transaction designed for the execution of automated logic on blockchains. Alas, smart contracts transactions are one of the major hindrances to blockchain throughput. Hence, improving the execution time of smart contracts is a prime challenge for Blockchains at large. To that end, concurrent execution of smart contract is an appealing direction, which has been adopted by several contemporary Blockchains like Solana, Aptos, Sui, Sei, and Monad.**

Executing smart contracts in parallel requires applying deterministic concurrency controls based on ensuring consistent ordering of all conflicting transactions in all miners/validators. Existing implementations rely on the Block's total ordering to resolve this requirement. Recently, it has been suggested that relying on minimal coloring of the conflict graph corresponding to the Block's transactions can provide a better performance potential, yet without any evaluation. In this paper, we compare between approaches to smart contracts parallelization. Our study[1] finds that in many situations, indeed the coloring-based ordering leads to significantly better performance than the Block order preserving approach. However, this gain has its limits, and it is not always guaranteed. In particular, the results are largely dependent on the conflict ratio in the conflict graph and the type of application.

***Index Terms*—Smart contracts, Parallel Execution, Graph Coloring, Deterministic Concurrency Controls, Blockchains**

I. Introduction

Smart contracts are the equivalent of general purpose programs in blockchains. That is, they enable enriching blockchains with general functionality beyond simple native direct asset transfer instructions. This is paramount in order to support more sophisticated financial transactions, and serves as a critical enabling technology behind the Web 3.0 vision.

However, the cost of smart contract execution has been identified as the main performance bottleneck in modern blockchains [3]. Given that modern CPUs have multiple cores, a natural approach to expedite smart contracts' execution time is by running them in parallel [4], [5], [6], [7], [8]. We note that in order to preserve the replicated state machine semantics [9] of blockchains, all transactions must be logically executed in the same deterministic total order. In other words, they must preserve *deterministic serializability*. In fact, the need for determinism is one of the significant differences between blockchains and traditional databases. Previous works on parallelizing smart contract execution have chosen to ensure that the serializability order by which transactions are logically executed obeys the order in which they appear within the block, to which we refer to as the *block order* (and between blocks, the consensus total order). This means that whenever two transactions conflict, their physical execution order must follow the order in which they were placed inside the block. In that sense, such approaches are *block order preserving*.

Recently, it has been shown in [10] that insisting on preserving the logical order within the block can lead to significant performance loss. They also showed that finding an optimal parallel deterministic serialization is equivalent to the graph coloring problem. Specifically, they suggest first computing a minimum coloring of the conflict graph that represents the block's transactions, and then ensuring that whenever two transactions conflict, their physical execution order obeys their respective colors numbers. Unfortunately, the work in [10] is purely theoretical, and to the best of our knowledge, has never been evaluated before. Further, minimum coloring is an NP-hard problem [11], and even its approximation is NP-hard.

In this paper, we test the performance gain that can be obtained by parallelizing smart contracts' execution, both using the traditional block order preserving approach [4], [5], [6], [7], [12], [13], [14], [15], [16], [17], [18], [19], [20], and when following the coloring approach of [10]. That is, we compare the running times of both parallelization approaches to each other and to serial execution. Further, given the NP-hardness of minimum coloring, we experiment with a few heuristic coloring algorithms and find that the greedy algorithm of [21] usually finds close to minimum coloring while its running time is very short compared to other transactions' execution costs. Hence, we are also the first to show that the coloring based approach can be made practical.

[1]Code and data are available online [1], [2].

For lack of credible blockchain specific benchmarks [3][2], we used the YCSB benchmark [22] from the BenchBase [23] repository in addition to creating artificial benchmarks. The artificial benchmarks are based on an e-commerce smart contract we developed and a wallet contract to cover a wide range of blockchain application scenarios [6], [24]. The wallet contract is essentially equivalent to the P2P transactions workload used for evaluation in [6]. These enabled us to explore the performance of the system under different workloads, different conflicts ratios, varying number of transactions per block, and varying the number of threads.

Our study finds that in most cases parallelization helps, and the coloring based approach can improve the latency of executing the entire block by a factor of up to 2.5. Yet, the level of improvement depends on the workload and the conflict ratio. Obviously, when the conflict ratio is 0%, the coloring based approach has no advantage over the block order preserving approach, yet both obtained a 15X improvement over serial execution in our tests (this exact number depends on the number of hardware cores in the machine and the number of threads used). Similarly, when the conflict ratio is 100%, there is no benefit for parallelism, and in fact can even be slightly slower due to its overheads. In between, even up to 80% conflict ratio gives significant improvement for the coloring based approach in most workloads we tested. For YCSB with a uniform distribution and an operation mix of 80% reads and 20% writes, the coloring-based approach using 28 threads achieved a 1.8X speedup over the block-ordering approach and a 3.5X speedup over serial execution.

Paper Roadmap: We start with a brief background in section II. Next, we introduce the concept of concurrent conflict graph schedulers, which is the basis for parallel execution of smart contracts in section III. We describe our implementation of all three methods in section IV, while the performance evaluation is described in section V. We survey related works in section VI and conclude with a discussion in section VII.

II. Background

A. Transactions and Conflicts

We refer to each smart contract invocation inside a block as a *transaction*. In particular, for simplicity, we assume that all transactions are the result of smart contract invocations, as is the case in many modern blockchains like Solana [25], Aptos [6], Sui [15], Sei [14], and ICP [26].

Each transaction may access various objects for both reading and writing. Formally, we assume some arbitrary block B with a set T of n transactions. Each transaction $tx \in T$ is a deterministic program with a specific list of inputs. The execution of tx may interact with the global state in addition to returning a result.

Using standard database terminology [27], we refer to the set of objects read by a given transaction as its *readset* and the set of objects written to by such a transaction as its *writeset*. We say that two transactions tx_1 and tx_2 are *conflicting* if either their writesets intersect, or the writeset of tx_1 (or tx_2, respectively) intersects with the readset of tx_2 (or tx_1, respectively). Here, conflicts are symmetric; therefore, they do not impose a directed dependency between two transactions.

A key observation made in [4], [28] is that executing each block in a deterministic and serializable manner is enough to guarantee correctness. The most common technique used for deterministic serializability is sequential execution, as described in Algorithm 1.

Algorithm 1 Sequential Scheduling

```
1: procedure ExecuteBlock(T: Transactions))
2:     [tx_1, ..., tx_n] ← deterministic order of txs in T
3:     for i=1,...,n do
4:         execute transaction tx_i
5:         emit transaction tx_i results
6:     return changes to global state
```

B. Transaction Reordering

It is known that a blockchain system can reorder transactions within the same block without violating strict serializability [4], [28]. However, as mentioned above, given the local independent execution of transactions among the different miners/validators of a blockchain, it is important that the logical serialization order of all transactions be the same. Previous works [7], [28], [6], [14], [15] relied on consensus and block ordering to enforce ordering among conflicting transactions. That is, whenever a pair of conflicting transactions tx_1 and tx_2 appear in this order in consensus and block ordering, they ensure that tx_1 will be executed, or at least committed, before tx_2.

In contrast, the authors of [10] proposed to color the *conflict graph*, which represents all conflicts among all transactions of a given block, and schedule transactions according to the color order. That is, for a given pair of conflicting transactions tx_1 and tx_2 such that the color of tx_1 is smaller than the color of tx_2, ensure that tx_1 is executed before tx_2. More precisely, each transaction only gets scheduled to run after all conflicting transactions with smaller colors have terminated. Unfortunately, the work of [10] is purely theoretical; hence, this work compares both approaches to transactions ordering.

III. Concurrent Conflict Graph Schedulers

Similarly to [4], [7], [28], [10], [6], we assume that for any two blocks B_i and B_j such that B_i is ordered before B_j in the consensus induced total order, then all transactions of B_i are executed before all transactions of B_j. Hence, we are left with scheduling transactions that appear in the same block. Recall that our goal is to enable parallelization of transaction execution, while preserving deterministic serialization.

A. Graph Scheduling

We follow the idea of *graph schedules* used in [7], [4], [12], [13], [28], [19], [4], [28], [10]: a graph schedule is a DAG of the transactions that respects conflicts [29]. A DAG is said

[2]While Diablo [3] suggests a benchmark, it does not really mimic object accesses and therefore not suitable for measuring the impact of parallelism.

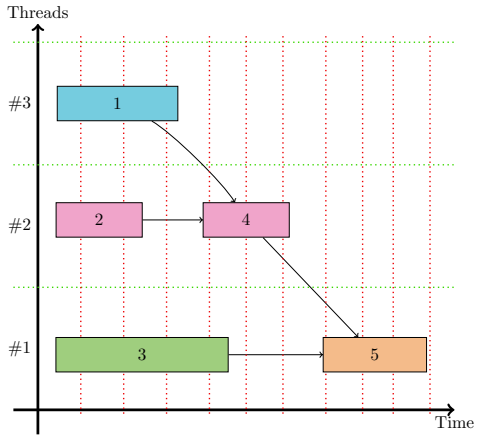


Fig. 1. Depiction of the execution of a graph schedule with 5 transactions and 3 hardware threads. The horizontal axis show the progress of time and vertical axis divides into the (concurrent) time lines of each hardware threads. The vertical dotted lines show the progress of each time unit.

to respect conflicts if it contains some directed path between any pair of conflicting transactions; thus, an (partial) ordering is implied between all conflicting transactions. Such orderings prevent the concurrent execution of conflicting transactions, which guarantees deterministic serializability.

A graph schedule is executed according to the DAG. Transactions are scheduled for execution immediately when there are no incoming edges in the DAG; transactions are removed from it when they complete. The scheduling of transactions repeats until none are left.

B. Optimal Scheduling

As originally proposed in [7] and adopted in [4], [12], [13], [28], [19], we start by describing the *block order preserving graph scheduler*, whose pipeline is illustrated in Figure 2a: (1) generate the conflict graph corresponding to the blocks transactions, (2) order the transactions according to the block order by directing the edges in the conflict graph accordingly, (3) thereby turning it into a DAG, (4) schedule each transaction as soon as it has no incoming edges in the DAG, (5) remove each transaction that terminates from the DAG.

In contrast, the work of [10] proposed to use the *coloring based graph scheduler*. This scheduler, illustrated in Figure 2b, uses the same steps as the block order preserving graph scheduler described above, except for replacing step 2 with: (2a) color the graph using an approximate minimal coloring algorithm, (2b) order the transactions according to the color order by directing the edges in the conflict graph accordingly.

C. Optimality and Graph Coloring

The work of [10] defined the latency of a schedule using the depth (maximal weight of vertexes in a path) of the corresponding scheduling graph. They have shown how to achieve the optimal latency using minimal graph coloring. In particular, they formally proved that when transactions are *homogeneous*(all have the same execution durations), coloring the conflict graph using a minimal coloring algorithm and then scheduling transactions according to their colors ordering yields optimal latencies. Yet, no actual implementation was presented in [10]. In this paper, we focus exclusively on blocks of homogeneous transactions. We note that ϵ-homogeneity is also relevant to the focus on homogeneous block. In ϵ-homogeneity, heterogeneous blocks are considered as homogeneous ones if the transaction lengths differ by no more than ϵ; however, the characteristics of ϵ are intentionally left out as they are closely related to the concrete use cases and to the ramifications of such an assumption.

Unfortunately, minimal vertex coloring is an NP-hard problem. In addition, it is known that minimal vertex coloring is hard to approximate; even finding a vertex coloring that is at most c times larger than the optimum is NP-hard *for any constant* $c > 1$ [30]. Hence, in this work we apply this approach with a heuristic coloring algorithm.

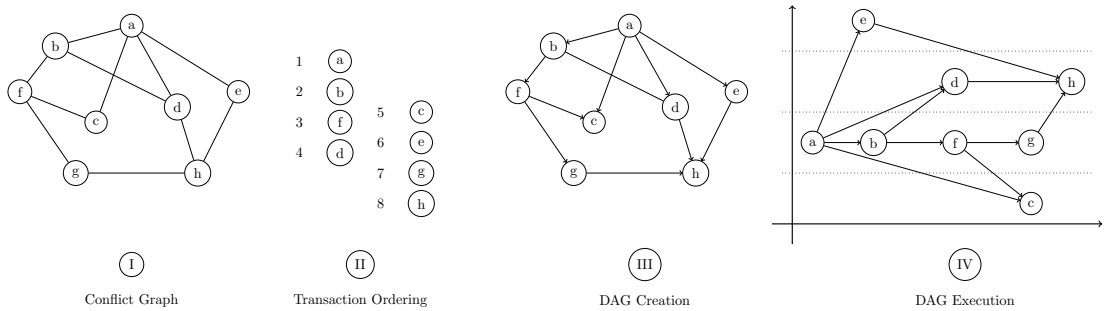
IV. Implementation

We have implemented sequential execution of blocks (aka Serial) and both parallel approaches, i.e., the block order preserving approach (aka Block Ordering) and the coloring based approach (aka Coloring). Figure 3 breaks down the stages of the execution pipeline.

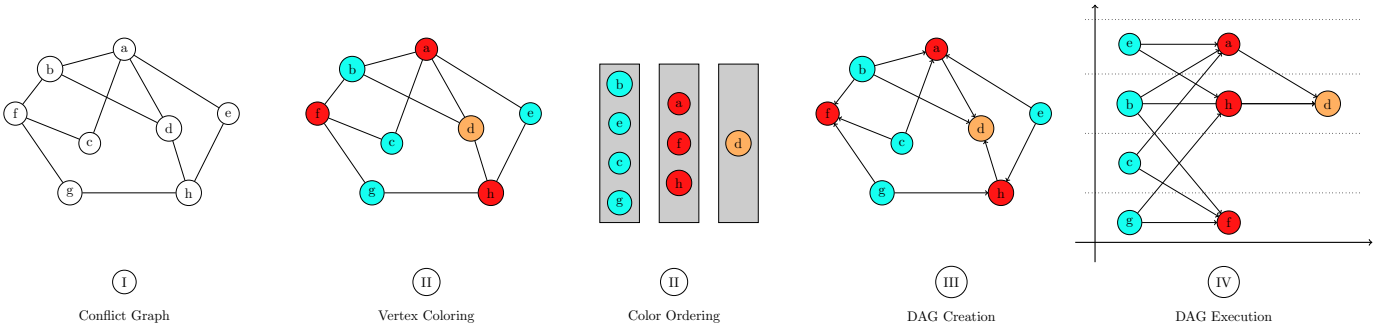
The *Parallel Conflict Graph Construction* Algorithm 2 is designed to efficiently identify conflicts among transactions in a block B by leveraging parallel processing. It begins by initializing an empty conflict graph $\bowtie$, which will represent the conflict relationships between transactions. The vertices of this graph is the set of transactions in the B. A global atomic counter is employed to facilitate dynamic, thread-safe assignment of transactions to multiple threads. Each thread executes a dedicated function, where it claims the next available transaction tx_i by incrementing the atomic counter. Once a transaction is claimed by a thread, it iterates through all other transactions tx_j in the block B to check for conflicts. Conflicts are determined by examining overlaps in addresses involved in *read-write*, *write-read*, or *write-write* conflicts. If a conflict is detected, a directed edge is added from tx_i to tx_j in the graph $\bowtie$. After all threads have completed their tasks, the procedure synchronizes and returns the fully constructed conflict graph $\bowtie$, which encapsulates all detected dependencies between transactions in B.

The algorithm presented in Algorithm 3 is a coloring-based scheduling method designed to color the conflict graph created in Algorithm 2 and generate a schedule T based on the colors of the nodes. Given the NP-hardness of minimal coloring [11], we have implemented the Greedy coloring technique [21] to color our conflict graph. During coloring, we also keep track of the number of nodes colored by each color category referred to as $colorCount$. Utilizing this, we design the schedule T. As shown in Algorithm 2 during the creation of schedule T, the edges are directed from transactions with higher $colorCount$ to transactions with lower $colorCount$.

We have also experimented with various parallel graph coloring algorithms [31], [32] but found that, for smaller scale graphs such as those used in blockchain (i.e., in the range of



(a) Order Preserving Graph Scheduling



(b) Coloring Graph Scheduling

Fig. 2. Comparison of graph scheduling

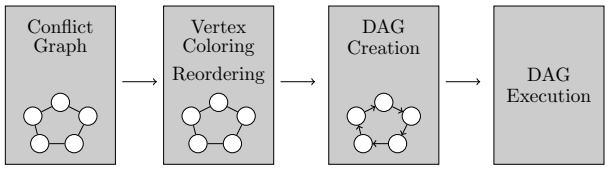


Fig. 3. The different stages in the pipeline in the implementation of the coloring based graph scheduling.

thousands of transactions), parallel coloring algorithms are not efficient enough. In fact, we observed that the serial greedy coloring algorithm performs best for both sparse and dense small graphs, offering optimal results in terms of computational efficiency and schedule creation.

V. Evaluation

A. Methodology

1) Metrics: We measure the time that it takes to complete the execution of an entire block of transactions, as a function of the percentage of conflicts, the number of threads, and the total number of transactions inside a block. The percentage of conflicts has an impact on the ability to parallelize the execution: if there are no conflicts, all transactions can be executed in parallel. In contrast, when all transactions conflict with each other, parallel executions degrade to completely sequential with an added overhead of synchronizing the concurrent threads. Hence, the interesting part is in the middle ground.

The number of threads is important since it translates to the maximal level of parallelism that the mechanism can utilize,

Algorithm 2 Parallel Conflict Graph Construction

```
1:  procedure ConflictGraph(Block B: {tx_1, tx_2, ..., tx_n})
2:      ⋈ ← InitializeGraph()  ▷ Initialize an empty conflict graph
3:      AtomicCounter ← 0                    ▷ Global atomic counter
4:      Create n threads
5:      for i = 1, ..., n do
6:          pThread(thread_i, ConflictDetection)
7:      threads.join()                       ▷ Wait for all threads to join
8:      return ⋈                             ▷ Return conflict graph
9:  procedure ConflictDetection
10:     while true do
11:         index ← AtomicIncrement(AtomicCounter)
12:         if index ≥ |B.Txns| then
13:             break
14:         tx_i ← B.Txns[index]
15:         for all tx_j = tx_{i+1}....tx_n do
16:             if (tx_i.ReadAddrs ∩ tx_j.WriteAddrs)
17:                 or (tx_i.WriteAddrs ∩ tx_j.ReadAddrs)
18:                 or (tx_i.WriteAddrs ∩ tx_j.WriteAddrs) then
19:                 AddEdge(⋈, tx_i, tx_j)   ▷ Add an conflict
20:     return
```

regardless of conflicts. On the other hand, once the number of threads surpasses the number of logical hardware cores, their ability to gain performance is limited, and we start hitting synchronization and context switching overheads.

The number of transactions per block impacts the time it takes to create the conflict graph, and in the coloring based approach, the time running time of the approximate

Algorithm 3 Coloring Based Scheduling

```
1:  procedure ColorSchedule(⋈, Block B)
2:      initialize available[n] ← false
3:      initialize colorCount[n] ← 0
4:      B[tx_1].color ← 0
5:      colorCount[0] ← colorCount[0] + 1

6:      for tx_i = tx_1, ..., tx_n do
7:          for tx_j = tx_1, ..., tx_n do
8:              if Edge(tx_i, tx_j) ∈⋈ & B[tx_j].color ≠ −1 then
9:                  available[B[tx_j].color] ← true
10:         for cr = 0, ..., n − 1 do
11:             if !available[cr] then
12:                 Break

13:         B[tx_i].color ← cr
14:         colorCount[cr] ← colorCount[cr] + 1

15:         for tx_j = tx_1 ..., tx_n do
16:             if Edge(tx_i, tx_j) ∈⋈ & B[tx_j].color ≠ −1 then
17:                 available[B[tx_j].color] ← false

18:     for all (tx_i, tx_j) ∈⋈ do
19:         C_i ← B[tx_i].color
20:         C_j ← B[tx_j].color

21:         if colorCount[C_i] > colorCount[C_j] then
22:             Add (tx_i → tx_j) to T                ▷ schedule T
23:         else
24:             Add (tx_j → tx_i) to T
25:     return (⋈, T)
```

coloring algorithm. Our measurements are inclusive of all these overheads. The results shown are an average of 10 runs.

2) Baselines: We compare the two variants of parallel execution mentioned in this paper, namely the block order preserving approach and the coloring based approach against each other and against serial execution of all transactions in a block in the order in which they appear there. In the graphs, we denote the results corresponding to the above schemes by Block Ordering, Coloring, and Serial, respectively. In both parallel cases, we measure the overall time including creating the conflict graph, turning it into a dependence DAG, and then scheduling and executing. For the coloring based approach, we also account for the time it takes to color the conflict graph.

3) Workloads: To mimic real world behavior, we have created an e-commerce application as well as a p2p transfers benchmark similar to [6], [24], and a YCSB benchmark [33]. E-commerce is a simple application which mimics a shopping transaction in which one client pays for a single shopping cart. The objects accessed by the e-commerce smart contract are the purchased items whose stock counts are being updated.

P2p transfers, on the other hand, enables a customer to perform the following operations: deposit money into his/her wallet account, withdraw money from his/her wallet account, check the balance in the wallet account, transfer money from his/her wallet account to another. The customer is identified by a customer name and a corresponding public key. The wallet account balance is stored at an address, derived from SHA 512 hash of customer's public key and the wallet's transaction family name-space. In our experiments, we have only invoked the transfer method.

To study the impact of conflicts on execution times, we created blocks with varying numbers of transactions, each taken from the above smart contracts, and varying the percentage of conflicts. We used a uniform distribution to model conflicts in the e-commerce application. For the p2p transfer application, we applied three conflict generation strategies: Zipfian to Zipfian, uniform to Zipfian, and uniform to uniform. In the uniform model, to establish some X percentage of conflicting transactions, we consider all pairs of transactions and draw $X\%$ of them as conflicting.

For the Zipfian model, the Zipfian distribution is used to assign the popularity of objects that can be accessed by different transactions. We have experimented with a $\theta = 0.4$ parameter. Regardless of the parameter θ, each transaction draws the objects it accesses from the respective Zipfian distribution, until the number of conflicts relative to all possible pairs of transactions is roughly equal to the target $X\%$.

To simulate the typical execution time of smart contracts on blockchains, we introduced a 6 ms delay for each transaction. This value was derived from a performance analysis conducted on Hyperledger Fabric [34]. For comparison, the average smart contract execution time on Hyperledger Sawtooth is approximately 40 ms [19]. This deliberate delay is intended to compensate for the overheads the virtual machine and cryptography compared to a native code execution. We note that if the individual smart contract execution latency is close to 0, then obviously one can reach very high throughput even without parallelization. On the other hand, as this latency increases, we can naturally expect parallelism to have an even more profound impact on performance.

YCSB Benchmark: We evaluated the performance using the YCSB benchmark [33] under two different data distributions: Uniform and Zipfian (with a skew factor of 0.9). We have used the BenchBase repository [23] for generating the YCSB benchmark operations. BenchBase is a multi-DBMS SQL benchmarking framework that typically issues operations directly to the underlying database. In contrast, blockchain execution is organized in blocks rather than as individual operations. To align with this execution model, we modified BenchBase to generate YCSB transactions according to the specified parameters and aggregate them into blocks. We configured the YCSB workload to include only read-modify-write and read operations, as these operations are most representative of blockchain transaction patterns.

4) Setup: All tests were executed on an Intel® Xeon® E5-2690 v4 @ 2.60GHz machine, with 56 logical cores, 28 physical cores (14 per socket). The hardware cache sizes are: 32KB L1d, 32KB L1i, 256KB L2, and 35840KB L3, and the total DRAM size is 164GB. The operating system is Linux Ubuntu 18.04.6 LTS. The code is written in C++ 11 and was compiled with gcc-11.1.0.

B. Results

1) Average Execution Time: We conducted a series of experiments to extensively evaluate the performance of our proposed framework under diverse scenarios, organized into three experimental setups. In the first experiment, we varied the number of transactions per block to analyze its impact on execution time, with the number of threads fixed at 56 and the conflict ratio set to 40 and 10 for e-commerce and p2p smart contracts respectively, and a 20% write percentage for YCSB. The results, presented in Figures 4, 7, 16. Figures 18 and 22, indicate that the execution time for all three implementations (Serial, Block Ordering, and Coloring) increases as the number of transactions per block grows. However, the rate of increase for Block Order and Coloring is significantly slower than that of Serial, with Coloring exhibiting the slowest growth rate.

In the second experiment, we vary the conflict ratio while keeping the number of threads constant at 56 and the number of transactions per block fixed at 2000. For YCSB, we have varied the percentage of writes among the operations. As expected, Figures 5, 8, 19 and 23 show that the sequential execution time remains roughly constant. However, the execution times for Block Ordering and Coloring increase exponentially with increasing conflict ratio. In case of YCSB, the growth for Block Odering and Coloring is linear with write percentage. At a conflict ratio of 100 (see Figure 5), all three algorithms exhibit similar execution times. This indicates that the overhead associated with conflict detection in both algorithms and the coloring time in Coloring is significantly lower than the time spent on transaction execution. Moreover, Figures 5 and 8 highlight that the growth rate for Coloring execution time is lower than that of Block Ordering, demonstrating the advantage of reordering transactions within a block over adhering strictly to block order.

In the third experiment, we analyze how varying the number of threads impacts execution time while keeping the number of transactions per block fixed at 2000 and the conflict ratio constant (40 for e-commerce and 10 for p2p). The sequential execution time remains unaffected by the number of threads. Figures 6, 9, 17, 20 and 24 illustrate that, for Block Ordering, the execution time decreases with increasing thread count up to 28 threads but starts to increase beyond this point, indicating contention associated with higher parallelism. In contrast, Coloring continues to exhibit performance gains until 56 threads, after which its execution time also begins to rise.

We also analyzed the impact of varying the read/write ratio and the number of transactions per block on the number of colors in YCSB benchmarking, illustrated in Figures 21a, 21b, 21c and 21d. An interesting observation is that the number of colors increases with the percentage of writes, even when the total number of transactions remains constant, for both uniform and Zipfian distributions. A higher number of colors indicates a greater number of transactions with increasing inter-transaction dependencies.

This behavior highlights the better scalability characteristics of Coloring. However, at extremely high thread counts, both algorithms face increased overhead due to our experimental setup being limited to 56 logical cores, which negatively impacts their performance.

2) Execution Time Breakdown: In this section, we show the performance breakdown of the various parallelization approaches, corresponding to their stages as illustrated in Figure 3. The results for the various parameters are listed in Table I. As shown, the approximate coloring algorithm is fairly fast, but as expected, depends on the number of transactions per block in a super-linear manner. Yet, even with 3,500 transactions per block, it takes less than 0.25 second, much less than the time it saves for most values of conflict ratios.

Also, DAG creation is a relatively major time consuming activity, especially as the number of transactions per block increases. This part is shared by both parallelization approaches, which is why the time spent in this activity is the same for both Block Ordering and Coloring. Interestingly, the DAG creation time initially grows with the number of conflicts, but then drops at 100% conflicts, since the graph becomes easy to compute when everyone is connected to everyone. Also, since our DAG creation protocol is concurrent, its time drops with the increase in the number of threads, until reaching the number of hardware virtual cores, which is then expected. Overall, we can conclude that improving DAG creation is an important aspect for future research, which could make both parallel approaches even more attractive.

VI. Related Work

Parallel BFT: To the best of our knowledge, the first work to consider parallel execution of Byzantine tolerant replicated state machine is [7]. The CBASE architecture of [7] places a parallelizer between the consensus layer and the multithreaded execution layer. Using the transactions read/write-sets, the parallelizer generates a conflicts graph and turns it into a DAG according to the transactions' ordering. Execution threads execute the transactions according to the DAG.

Parallelizing Blockchains Research: Dickerson et al. [5] introduced the concept of parallel execution of Ethereum [35] transactions using Software Transactional Memory (STM). In this approach, the block producer executes transactions using STM, allowing the discovery of a serializable concurrent state. This state is then shared with validators to achieve deterministic execution. Following this, multiple STM-based concurrent transaction execution frameworks for blockchains have been developed [4], [6], [12], [13], [20], [28].

Saraph et al. [8] introduced an algorithm for the Ethereum blockchain that categorizes transactions into two distinct groups: concurrent and sequential, based on their read and write sets. Additionally, a case study on Hyperledger Sawtooth demonstrated a DAG-based parallel scheduler that uses the read and write address information of transactions to create a concurrent schedule for execution [19]. Recent research has explored the lower bounds on the maximal potential attainable parallelism in the execution of smart contract transactions [10]. It has been shown that adhering to the consensus total order for

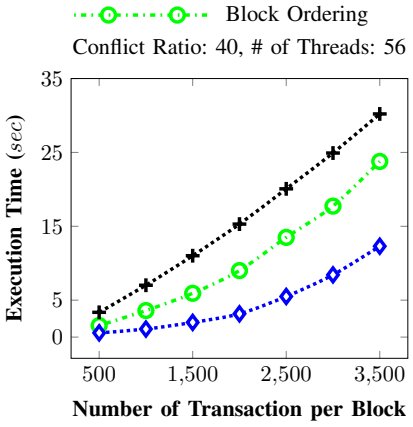


Fig. 4. E-Commerce: Experiment 1

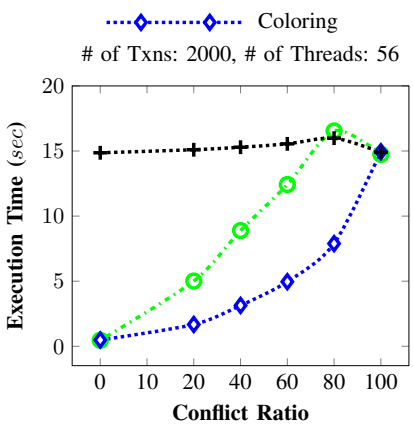


Fig. 5. E-Commerce: Experiment 2

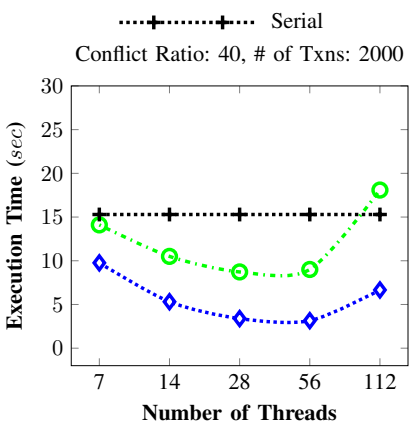


Fig. 6. E-Commerce: Experiment 3

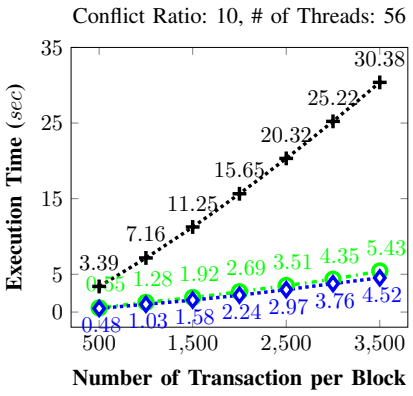


Fig. 7. P2P (Zipfian-Zipfian): Experiment 1

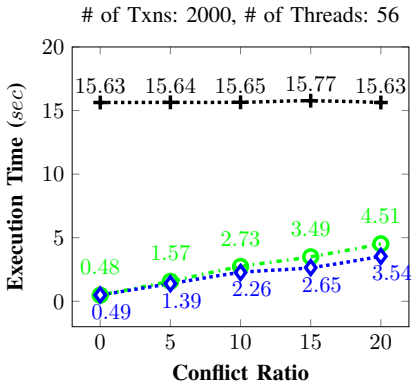


Fig. 8. P2P (Zipfian-Zipfian): Experiment 2

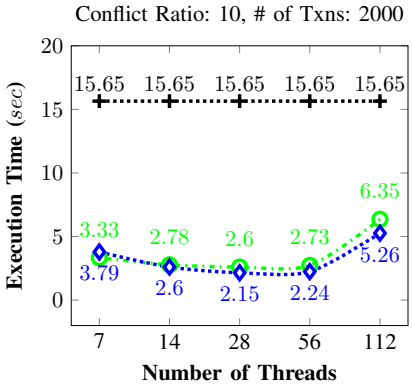


Fig. 9. P2P (Zipfian-Zipfian): Experiment 3

E-commerce: Transactions: 2000, Conflicts: 40% Time (seconds)										
	Block Ordering			Coloring						Serial
Threads	DAG Create	Execute	Total Time	DAG Create	Coloring	Reordering	Execute	Total Time	Colors	Total Time
7	7.1437	6.9605	14.1042	7.2043	0.1020	0.0238	2.4296	9.7597	195	15.2921
14	3.6617	6.8397	10.5013	3.6357	0.0806	0.0163	1.5817	5.3144	195	15.2921
28	1.8825	6.8397	8.7222	1.8608	0.0787	0.0101	1.4368	3.3863	195	15.2921
56	1.5000	7.5048	9.0048	1.5051	0.0797	0.0091	1.5533	3.1471	195	15.2921
112	1.5218	16.5671	18.0889	1.5023	0.0807	0.0098	5.0682	6.6610	195	15.2921

TABLE I
EXECUTION TIME BREAKDOWN FOR E-COMMERCE APPLICATION; VARYING THE NUMBER OF THREADS

the serialization order of transactions can significantly reduce parallelism, while other deterministic ordering approaches, such as minimal graph coloring, offer greater potential.

In the work of [36], only the block proposer computes the dependency graph, which is sent along with the block to the validators. They presented some heuristics on how the block creator can increase the concurrency of the dependency graph and reduces the number of aborts. Such a model suffers from high communication overhead, which is partially handled in [36]. Further, the results of [36] only fit permissioned blockchains, while the execution phase at the validators need to be tightly coupled into the BFT consensus protocol.

Existing Parallel Blockchains: Several blockchains already apply parallelism to smart contracts execution. Some. like Solana [25], do so using pessimistic approaches, while others, like the Block-STM from Aptos [6], Sui from Mysten-Labs [15], Sei V.2 [14], and Monad [37], apply optimistic concurrency control. The latter reduces the synchronization overhead, and may be able to exploit parallelization better, but suffers from aborts and the need to re-execute transactions when there are conflicts. In both cases, the serialization order must obey the total order in which transactions were placed in the block, and the consensus order across blocks. However, as been shown in [10], this order can greatly impact the actual parallelism level and the number of aborts and re-executions, and therefore in this work we study how to optimize this order.

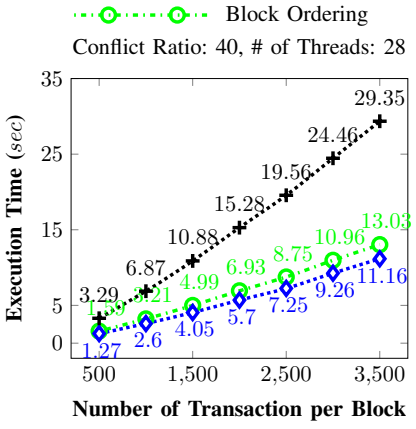


Fig. 10. P2P (Uniform-Zipfian): Experiment 1

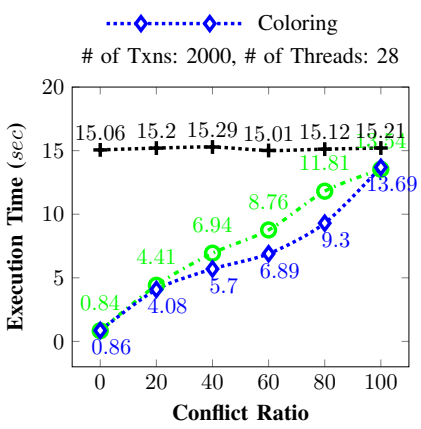


Fig. 11. P2P (Uniform-Zipfian): Experiment 2

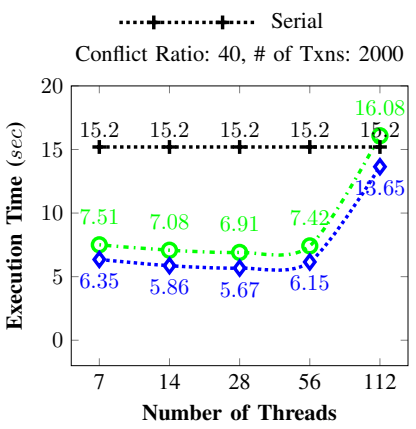


Fig. 12. P2P (Uniform-Zipfian): Experiment 3

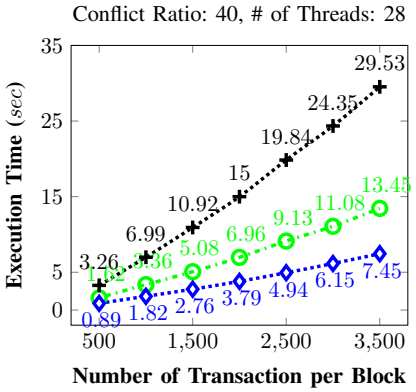


Fig. 13. P2P (Uniform-Uniform): Experiment 1

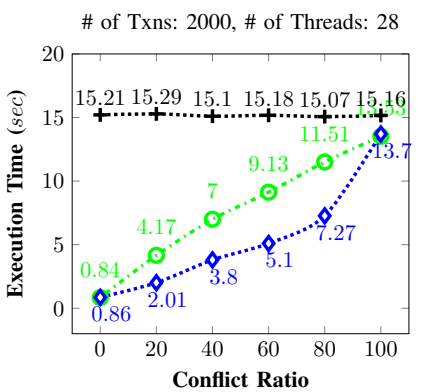


Fig. 14. P2P (Uniform-Uniform): Experiment 2

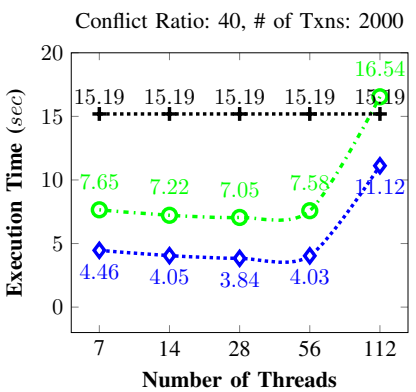


Fig. 15. P2P (Uniform-Uniform): Experiment 3

E-commerce: Transactions: 2000, Threads: 56 Time (seconds)										
%	Block Ordering			Coloring						Serial
Conflicts	DAG Create	Execute	Total Time	DAG Create	Coloring	Reordering	Execute	Total Time	Colors	Total Time
0	0.1878	0.2763	0.4640	0.1845	0.0106	0.0034	0.2837	0.4822	1	14.8643
20	0.7436	4.2656	5.0092	0.7357	0.0459	0.0051	0.8875	1.6742	112	15.0979
40	1.5060	7.3810	8.8870	1.4997	0.0804	0.0083	1.5465	3.1349	195	15.2921
60	2.4702	9.9528	12.4231	2.4661	0.0985	0.0115	2.3827	4.9588	299	15.5495
80	3.8657	12.6747	16.5404	3.8787	0.1131	0.0141	3.8809	7.8867	462	15.9891
100	0.3124	14.4289	14.7413	0.3103	0.1381	0.0158	14.4774	14.9417	2000	14.9037

TABLE II
EXECUTION TIME BREAKDOWN FOR E-COMMERCE APPLICATION; VARYING THE CONFLICT RATIO

Hyperledger and Derivatives: In Hyperledger fabric [38], transactions are first being simulated by the transaction initiator, and then sent to be totally ordered, then verified, and if successful, also being committed. This enables inherent parallelism, as each transaction is potentially simulated independently of the others. However, this also leads to a large number of aborts. The work of [39] reduces the number of aborts by reordering transactions such that writing transactions are delayed after concurrent transactions that read from the same variables. Yet, the optimizations reported in [39] are only applicable to Hyperledger-like blockchains.

Network Level Parallelism: While these solutions work on improving the throughput in a single node of the blockchain, some have focused on scalability improvement through optimization at a network level. The sharding technique involves splitting network nodes into smaller groups, allowing them to process transactions in parallel with one another [40]. Sidechains are another solution where separate blockchains are linked to the main blockchain, allowing for specific functionalities without congesting the main chain [41].

Workloads: Most works that study blockchains' performance rely on artificial workloads. To enable fair comparison of raw Blockchain performance, a standardized synthetic benchmark called Diablo [3] was proposed. Yet, Diablo does not focus on parallel execution. The transactions in Diablo are designed

E-commerce: Conflicts: 40%, Threads: 56 Time (seconds)										
TXs per Block	Block Ordering			Coloring						Serial
	DAG Create	Execute	Total Time	DAG Create	Coloring	Reordering	Execute	Total Time	Colors	Total Time
500	0.0490	1.5165	1.5655	0.0492	0.0058	0.0025	0.5136	0.5711	77	3.3529
1000	0.2583	3.3327	3.5910	0.2617	0.0174	0.0031	0.8076	1.0898	116	7.0192
1500	0.7852	5.1430	5.9282	0.7844	0.0475	0.0055	1.1493	1.9867	155	11.0164
2000	1.5000	7.5048	9.0048	1.5051	0.0797	0.0091	1.5533	3.1471	195	15.2964
2500	3.6486	9.8494	13.4980	3.4511	0.1170	0.0110	1.9200	5.4990	226	20.0723
3000	5.8743	11.8545	17.7288	5.8650	0.1716	0.0153	2.3730	8.4248	262	24.9171
3500	9.2334	14.5440	23.7774	9.2361	0.2219	0.0182	2.8268	12.3030	292	30.2099

TABLE III
EXECUTION TIME BREAKDOWN FOR E-COMMERCE APPLICATION; VARYING THE NUMBER OF TXS PER BLOCK

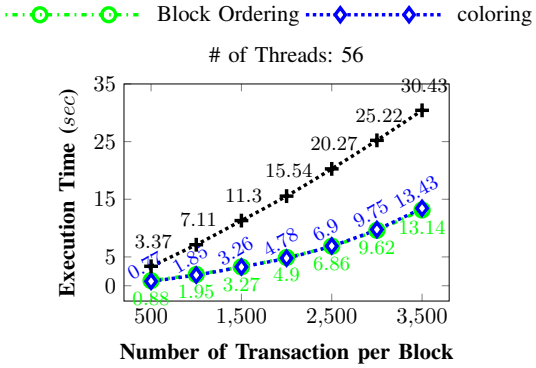


Fig. 16. Mixed: Experiment 1

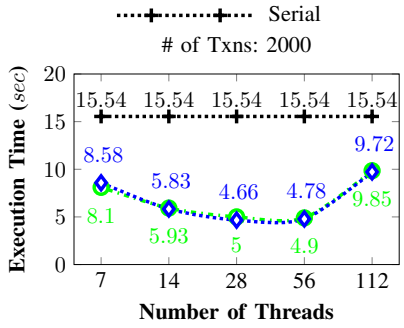


Fig. 17. Mixed: Experiment 3

to mimic the raw computational overhead for real world applications; however, these are not actual transactions but rather computational tasks. In order to test parallelization, some works, e.g., [4], [28] utilize database inspired workloads such as YCSB [33], TPC-C [42] and TPC-E [43].

VII. DISCUSSION & CONCLUSION

In this paper, we have explored the benefits of parallelizing smart contracts execution in terms of reducing the overall latency of processing all transactions inside a given block. We have considered two main approaches for ensuring deterministic serializability of the executed transactions, which is required to ensure correctness. The first method, the block order preserving approach, is the one currently used by several blockchains that support concurrent executions [4], [28], [15], [14], [6], which requires that conflicting transactions are ordered in the order they appear in the block. The second method, the coloring based approach, which was recently proposed [10] but never tested before, apply (approximate) minimal coloring to the respective conflict graph, and use the color order to order conflicting transactions.

We have studied the performance of these parallelization approaches compared to serial execution, in a variety of settings that differ in the number of threads, the conflict ratio between transactions in the same block, and the number of transactions inside the block. Our measurements revealed that parallelization usually helps, and the coloring based approach is also often superior to the block order preserving approach. However, this is not unconditional, and depends on specific values of the above parameters. When there are no conflicts, coloring cannot produce better parallelization, and only increases pre-processing overheads. At the other extreme of 100% conflicts, all transactions must follow each other. In between, the coloring based approach is usually better, but the improvement level depends on the conflict graph that is derived from the workload, which impacts the chromatic number. For example, we have witnessed that for the e-commerce application, coloring based ordering can yield up to 2.5 faster runs than preserving the block order, but for the p2p application the improvement is only 5-30% under the same parameters, and in a few situations, the block order approach is even slightly better. This is related to the conflict graphs generated by the p2p application, and their relatively high number of colors compared to the e-commerce application. As a future direction, one can think of a hybrid approach, in which based on the number and types of conflicts, the algorithm can decide whether to perform coloring or rely on the block order. Our results provide guidelines for such an effort.

As in many other efforts to parallelize the execution of smart contracts [4], [28], [15], [14], [25], we assume knowledge of the readsets and writesets. Such knowledge can be obtained either by explicit decelerations, code analysis, or tentative execution. Tentative execution of transactions is often performed in any case in order to calculate the amount of gas that the transaction should be provided with. Many existing blockchains either enable or require such decelerations, e.g., [44]. Even when not mandated, users are encouraged to declare the objects they intend to use through lower gas fees

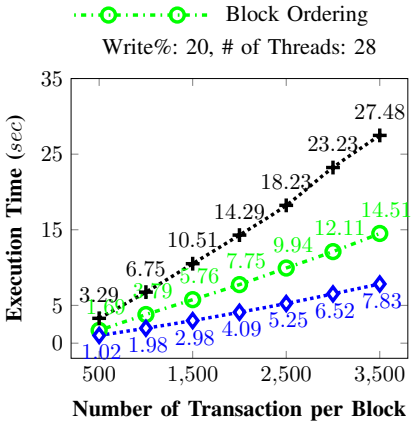


Fig. 18. YCSB Uniform: Experiment 1

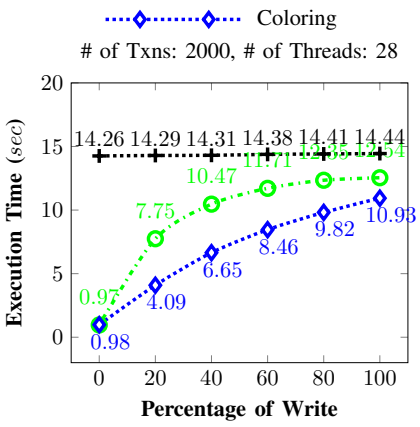


Fig. 19. YCSB Uniform: Experiment 2

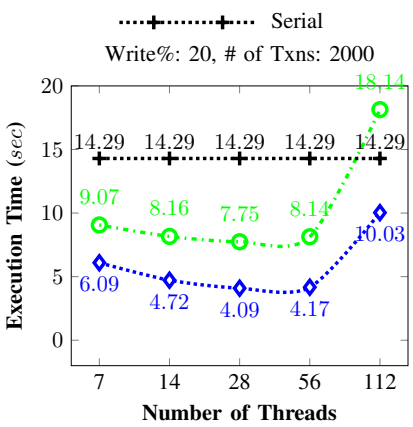
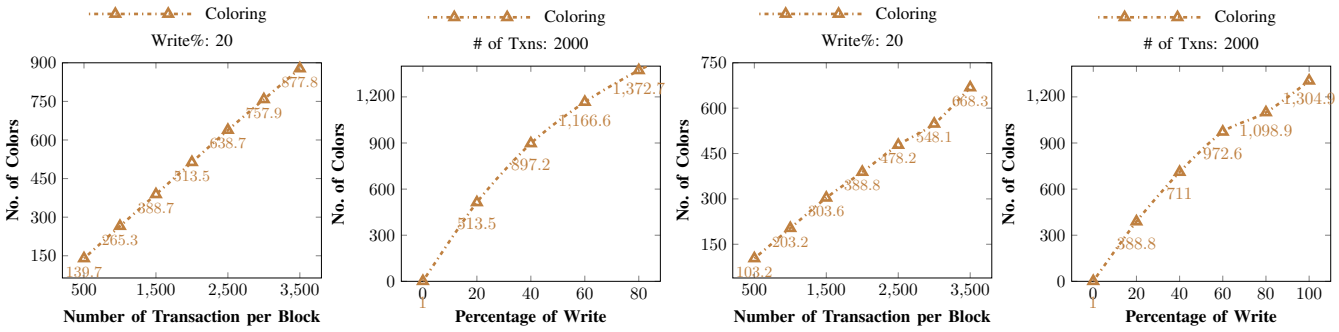


Fig. 20. YCSB Uniform: Experiment 3



(a) Uniform: Experiment 1 (b) Uniform: Experiment 2 (c) Zipfian: Experiment 1 (d) Zipfian: Experiment 2

Fig. 21. YCSB Number of Colors

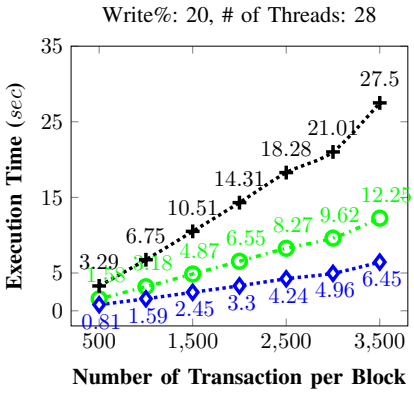


Fig. 22. YCSB Zipfian: Experiment 1

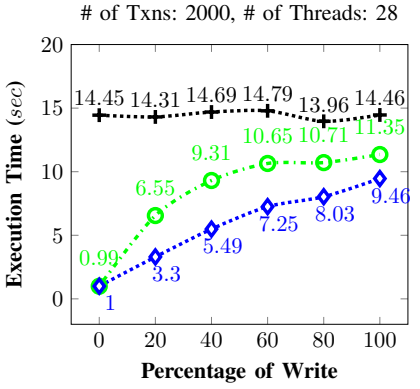


Fig. 23. YCSB Zipfian: Experiment 2

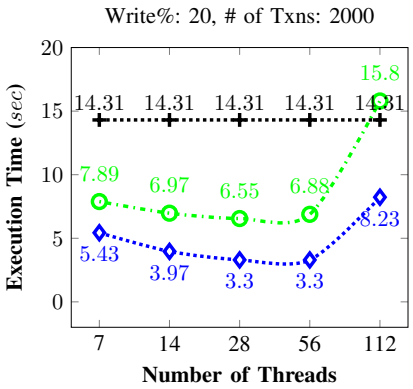


Fig. 24. YCSB Zipfian: Experiment 3

when accessing only declared objects. Such decelerations are helpful for multiple runtime optimizations of the blockchain's virtual machine, and not only for parallelization.

As mentioned, minimum coloring is an NP-hard problem [11] and is even hard to approximate. The greedy coloring algorithm we used [21] seemed to strike a good balance between its own running time and the quality of the coloring it returned.

Interestingly, the heuristic coloring algorithm did not impose significant overheads compared to the time it usually saved for the execution phase. However, a major performance bottleneck for both parallel approaches turned out to be the parallel DAG construction. Improving it is left for future work.

Acknowledgments: This work is partially funded by a grant from DST, GoI CRG/2022/009391 and the Ethereum Foundation, as part of the 2025 Academic Grants Round.

REFERENCES

[1] P. Authors. (2025) Code material. [Online]. Available: https://github.com/PDCRL/Coloring-SCT-PRDC2025.git

[2] ——. (2025) Data files. [Online]. Available: https://github.com/PDCRL/Coloring-SCT-PRDC2025.git

[3] V. Gramoli, R. Guerraoui, A. Lebedev, C. Natoli, and G. Voron, "Diablo: A Benchmark Suite for Blockchains," in *Proc. of ACM EuroSys*, 2023, p. 540–556.

[4] M. J. Amiri, D. Agrawal, and A. El Abbadi, "ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems," in *IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1337–1347.

[5] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding Concurrency to Smart Contracts," in *Proc. of ACM Principles of Distributed Computing (PODC)*, 2017, pp. 303–312.

[6] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing," in *Proc. of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2023, p. 232–244.

[7] R. Kotla and M. Dahlin, "High Throughput Byzantine Fault Tolerance," in *Int. Conference on Dependable Systems and Networks (DSN)*, 2004, pp. 575–584.

[8] V. Saraph and M. Herlihy, "An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts," ser. Proc. of Tokenomics, 2019.

[9] F. B. Schneider, "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[10] Y. Hay and R. Friedman, "Batch-Schedule-Execute: On Optimizing Concurrent Deterministic Scheduling for Blockchains," in *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2024.

[11] R. M. Karp, *Reducibility Among Combinatorial Problems*. Springer, 1972, pp. 85–103.

[12] P. S. Anjana, H. Attiya, S. Kumari, S. Peri, and A. Somani, "Efficient Concurrent Execution of Smart Contracts in Blockchains Using Object-Based Transactional Memory," in *Networked Systems*, C. Georgiou and R. Majumdar, Eds. Springer International Publishing, 2021, pp. 77–93.

[13] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts," in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 83–92.

[14] Sei, "Sei v2 - The First Parallelized EVM Blockchain," https://blog.sei.io/sei-v2-the-first-parallelized-evm/.

[15] S. Foundation, "All About Parallelization," https://blog.sui.io/parallelization-explained.

[16] d. Miller, H. F. Korth, and R. Palmieri, "POSTER: OCToPus: Semantic-aware Concurrency Control for Blockchain Transactions," in *Proc. of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024, p. 463–465.

[17] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, and R. Palmieri, "Processing Transactions in a Predefined Order," in *Proc. of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019, p. 120–132.

[18] G. Mitenkov, I. Kabiljo, Z. Li, A. Spiegelman, S. Vusirikala, Z. Xiang, A. Zlateski, N. P. Lopes, and R. Gelashvili, "Deferred Objects to Enhance Smart Contract Programming with Optimistic Parallel Execution," 2024, arXiv #2405.06117.

[19] M. Piduguralla, S. Chakraborty, P. S. Anjana, and S. Peri, "DAG-Based Efficient Parallel Scheduler For Blockchains: Hyperledger Sawtooth a Case Study," in *Proc. of the 29th European Conference on Parallel and Distributed Computing (EuroPar)*, 2023, p. 184–198.

[20] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "OptSmart: a Space Efficient Optimistic Concurrent Execution of Smart Contracts," *Distributed and Parallel Databases*, May 2022.

[21] A. Kosowski and K. Manuszewski, "Classical Coloring of Graphs," in *Graph Colorings and Bernoulli Subflows*. American Mathematical Society, 2004, pp. 2–19.

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[23] cmu-db, "Benchbase: Multi-dbms sql benchmarking framework via jdbc," https://github.com/cmu-db/benchbase, 2025, accessed: 2025-08-09.

[24] Askmish, "Sawtooth-Simplewallet," https://github.com/askmish/sawtooth-simplewallet.

[25] A. Yakovenko, "Sealevel — Parallel Processing Thousands of Smart Contracts," https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192.

[26] Dfinity, "ICP Execution Layer," 2025. [Online]. Available: https://learn.internetcomputer.org/hc/en-us/articles/34208985618836-Execution-Layer

[27] H. Garcia-Molina, J. Ullman, and J. Widom, *Database Systems: The Complete Book, 2nd Edition*. Pearson, 2008.

[28] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database," *Proc. of the VLDB Endowment*, vol. 12, no. 11, p. 1539–1552, Jul. 2019.

[29] I. A. Escobar, E. Alchieri, F. L. Dotti, and F. Pedone, "Boosting Concurrency in Parallel State Machine Replication," in *Proc. of the 20th ACM/IFIP International Middleware Conference*, 2019, p. 228–240.

[30] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," *Theory of Computing*, vol. 3, no. 6, pp. 103–128, 2007. [Online]. Available: https://theoryofcomputing.org/articles/v003a006

[31] M. T. Jones and P. E. Plassmann, "A Parallel Graph Coloring Heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[32] N. Singhal, S. Peri, and S. Kalyanasundaram, "Practical Multi-threaded Graph Coloring Algorithms for Shared Memory Architecture," in *Proc. of the 18th International Conference on Distributed Computing and Networking*, ser. ICDCN. ACM, 2017.

[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010, p. 143–154.

[34] Q. Nasir, I. A. Qasse, M. Abu Talib, and A. B. Nassif, "Performance Analysis of Hyperledger Fabric Platforms," *Security and Communication Networks*, vol. 2018, no. 1, 2018.

[35] V. Buterin, "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform," 2013.

[36] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A High Performance Concurrency Protocol for Smart Contracts of Permissioned Blockchain," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 11, pp. 5070–5083, 2022.

[37] M. Labs, "Parallel Execution," https://docs.monad.xyz/technical-discussion/execution/parallel-execution.

[38] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: a Distributed Operating System for Permissioned Blockchains," in *Proc. of the 13th ACM EuroSys Conference*, Apr. 2018.

[39] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric," in *Proc. of the ACM International Conference on Management of Data*, ser. SIGMOD, 2019, p. 105–122.

[40] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards Scaling Blockchain Systems via Sharding," in *Proc. of the Int. ACM Conf. on Management of Data*, ser. SIGMOD, 2019, p. 123–140.

[41] A. Singh, K. Click, R. M. Parizi, Q. Zhang, A. Dehghantanha, and K.-K. R. Choo, "Sidechain Technologies in Blockchain Networks: An Examination and State-of-the-Art Review," *Journal of Network and Computer Applications*, vol. 149, 2020.

[42] T. P. P. Council. (1992) TPC-C: On-Line Transaction Processing Benchmark. [Online]. Available: https://www.tpc.org/tpcc/default5.asp

[43] ——. (2007) TPC-E: On-Line Transaction Processing Benchmark. [Online]. Available: https://www.tpc.org/tpce/default5.asp

[44] L. Heimbach, Q. Kniep, Y. Vonlanthen, R. Wattenhofer, and P. Züst, "Dissecting the EIP-2930 Optional Access Lists," url=https://arxiv.org/abs/2312.06574, 2023.