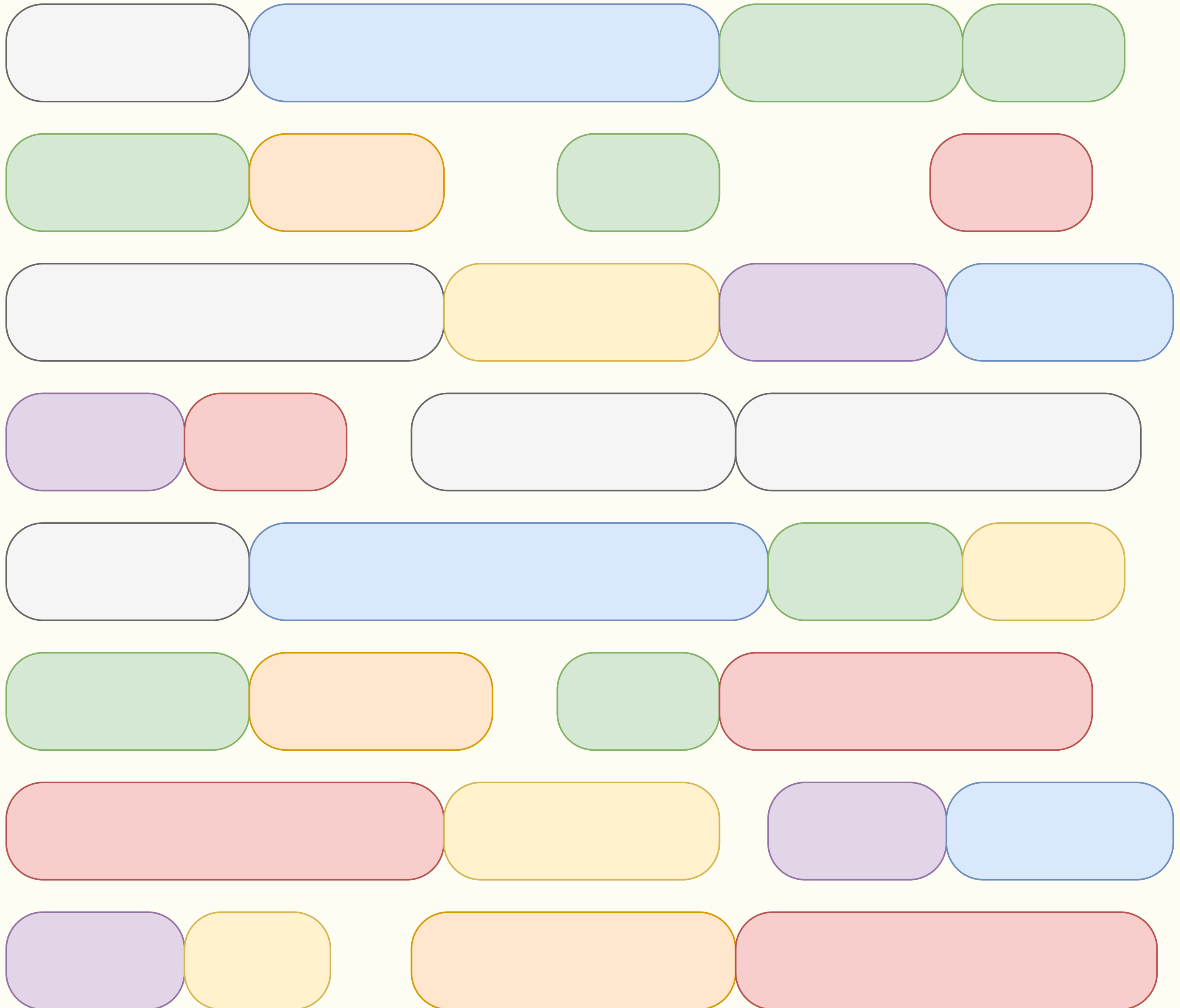
Optimizing Blockchain Transaction Execution Performance by Applying Genetic Sequencing

Artjom Pugatsov



Optimizing Blockchain Transaction Execution Performance by Applying Genetic Sequencing

by

Artjom Pugatsov

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended on April 21st, 2025

Supervisor:	Jérémie Decouchant, Delft University of Technology
Co-supervisor:	Can Umut Ileri, IOTA Foundation & Delft University of Technology
Faculty:	Electrical Engineering, Mathematics and Computer Science
University:	Delft University of Technology

Contents

I Introduction 1

II Background 2
II-A Heuristic ordering . . . 2
II-B Sequencing objectives . . . 3
II-C Sequencing requirements . . . 3
II-D Scheduling estimation . . . 3

III System Model and Objectives 4
III-A Assumptions . . . 4
III-B Single-shot scheduling . . . 5
III-C Continuous execution . . . 5
III-D Fair ordering . . . 6

IV Genetic Algorithm for Ordering 6
IV-A Genetic ordering . . . 7
IV-B Adding fair ordering to the genetic algorithm . . . 8

V Performance Evaluation 8
V-A Dataset and parameters . . . 8
V-B Sequencing baselines . . . 9
V-C Sustained congestion . . . 9
V-D Spike in transactions . . . 10
V-E Fair ordering . . . 11
V-F Robustness to execution time estimation errors . . . 12
V-G Throughput, latency and full summary . . . 13

VI Related Work 14
VI-A Sequencing on popular blockchains . . . 14
VI-B Fair ordering . . . 17
VI-C Combinatorial optimization for sequencing . . . 17
VI-D Execution . . . 18

VII Discussion 18

VIII Conclusion 18

References 19

***Abstract*—The successive generations of consensus algorithms progressively shifted the performance bottleneck of blockchains to the execution layer. Recent works have addressed this bottleneck by parallelizing the execution of transactions. Historically, transaction ordering was left to the discretion of validators, a practice that lacked transparency and gave rise to Maximal Extractable Value (MEV) attacks where transaction ordering is manipulated for private gain. More recently, the focus has shifted toward fair ordering protocols that prioritize chronological submission. However, fair ordering is often misaligned with validator incentives and negatively impacts execution throughput under high congestion. In this work, we address the tension between validator revenue and fair ordering using a dynamic optimization framework.**

We define a blockchain-independent model to evaluate transaction ordering in a continuous setting where the execution of successive blocks can overlap. Within this model, we propose an anytime genetic algorithm. We use real-world blockchain data and execution time estimates within realistic error margins, showing that this approach increases validator profit by around 15% and accelerates congestion relief. We also quantify the impact of adding fair ordering constraints on validator revenue during congestion, showing that revenue decreases by around 50%.

***Index Terms*—Blockchain, Execution layer, Sequencing, Byzantine Fault Tolerance**

I. Introduction

Blockchain systems have historically suffered from efficiency and throughput limitations. Bitcoin and Ethereum have estimated maximum transaction throughputs of 7 and 40 TPS, respectively [1, 2]. Meanwhile, blockchain applications have expanded to include use cases beyond traditional decentralized finance (DeFi) applications, such as the Internet of Things which require much greater throughput.

Blockchains typically operate in the Byzantine setting, where actors may be malicious, and are therefore required to solve the Byzantine Fault-Tolerant State Machine Replication (BFT-SMR) problem. Early blockchain designs followed a monolithic architecture, which complicated their optimization, as improvements to one part of the system would necessarily impact its entirety. This design limitation motivated the emergence of a new class of blockchain, known as lazy blockchains [3–5]. Lazy blockchains separate the various BFT-SMR steps, i.e., transaction dissemination, consensus, verification, and execution, which can therefore be independently optimized and processed concurrently. In such parallel systems, the overall performance is determined by the speed of the slowest component. As a result, most of the initial efforts focused on optimizing blockchain consensus, as it has historically been the most inefficient component [6]. However, recent directed acyclic graph (DAG)-based consensus algorithms brought a major increase in throughput [7–10], and execution has become the performance bottleneck of lazy blockchains.

There are two approaches for optimizing the execution stage: vertical and horizontal. Vertical optimization aims at optimizing individual transaction execution. It is highly dependent on the types of transactions, making it less generalizable. An example of such an approach is the analysis of smart contract programming languages [11, 12]. Horizontal optimization focuses on the ability to execute multiple transactions in parallel. This approach is more generalizable, as it abstracts transaction execution. It has been the dominant research focus for execution optimization. The general approach is to split the transactions based on their interdependence and distribute them across multiple independent executors [13–19]. Such an approach has led to significant throughput improvements with the state-of-the-art execution engines reportedly reaching 170k TPS [19].

The key property that enables efficient horizontal parallelization is transaction independence. As each transaction can be seen as a list of read and write operations on some set of accounts or objects, two transactions are independent if both transactions perform read-only operations on their overlapping objects. Independent transactions can be executed concurrently on different machines, requiring only a final state merge. Saraph and Herlihy [20] analyzed historical Ethereum transaction data and found that such parallelization could double execution speed. However, the authors also highlight the limitations of direct parallelization: a few transactions are responsible for causing major interdependence between transactions. Eliminating these problematic transactions would allow performance to increase by a factor of 8 compared to the sequential execution schedule. However, how these problematic transactions are identified and handled depends on the ordering policy applied before execution - a design decision that current blockchain systems leave largely unspecified.

Traditional blockchains such as Ethereum or Bitcoin, which execute transactions sequentially, allow validators to determine the transaction order themselves. Initially, the idea was to enable congestion control indirectly through self regulation by allowing users to submit gas prices, incentivizing validators to include higher-paying transactions first. The primary aim of such a mechanism was to handle the congestion resulting from spam by introducing a barrier to creating a transaction [21]. Although effective at filtering noise, this approach has significant drawbacks when handling congestion stemming from a genuine lack of execution resources. This approach to congestion management does not determine a precise ordering, leaving validators to arbitrarily reorder transactions. In practice, this freedom is often exploited in a non-transparent way to extract additional value for validators in a phenomenon known as Maximal Extractable Value (MEV) [22]. This lack of ordering transparency has given rise to research into fair ordering [23, 24]. The fair ordering approach gives up profit in favor of matching the transaction arrival time as closely as possible. These two paradigms exist at the opposite ends of a spectrum: unconditional validator ordering lacking any transparency on one end, and overly rigid ordering with total disregard for validator interests on the other.

In this work, we explore a middle ground between these two extreme approaches, building on the transaction sequencing approach as applied by Sui [25, 26]. In this framework, transactions are initially ordered and then some of them are deferred to the next block based on their resource usage in case

of congestion. In its current form, Sui first orders transactions by gas price, then evaluates them sequentially to determine which transactions to defer based on their predicted execution time and resource usage. A transaction is deferred to the next sequencing round if the cumulative estimated execution time of transactions touching the same object would exceed a protocol-defined capacity limit. Execution time estimates and object conflicts are only used in the deferral step. We expand on this idea by incorporating execution time estimates and object conflict information into the initial ordering step. Then we extend this concept into a continuous execution model where execution of successive blocks can overlap. We do this by applying an anytime genetic algorithm that can be dynamically adjusted based on congestion level. We evaluate it for validator profit on real-world Sui data and find that it improves validator profit by around 15% compared to the baseline and helps the congestion clear faster. We also extend this approach to include fair ordering by moving from total order to causal order and quantifying the profit-fairness tradeoff, finding that adding fair ordering reduces the profit by around 50%.

In summary, this paper makes the following contributions:

- We formalize a blockchain independent model to evaluate different ways of sequencing transactions in a continuous execution scenario.
- We extend this model to account for the fair ordering constraint.
- We propose and implement several heuristic sequencing baselines and a genetic algorithm for sequencing, which takes into account transaction execution time estimates and object conflict information.
- We evaluate the performance of different sequencers on real-life Sui data and synthetic Ethereum-based data by simulating sequencing and execution steps.
- We repeat all experiments under realistic perturbations of the predicted execution times.
- We find that applying genetic ordering increases validator profit by around 15%.
- We quantify impact of fair ordering on validator profit, showing that it reduces revenue by around 50%.
- Our code is publicly available online at https://github.com/Artjom-Pugatsov/genetic-sequencing.

II. Background

In this work, we focus on the sequencing layer of a blockchain. We define the sequencing layer as a part of the blockchain responsible for two operations: ordering of transactions and congestion control. Currently, many blockchains do not have an explicit sequencing step. In such cases, consensus takes on the responsibilities of the sequencing step by determining transaction order through block building. Our definition of the sequencing layer is primarily informed by Suis architecture, with consensus only responsible for producing a set of agreed-upon transactions without ordering them. Then the sequencing layer produces an ordering of transactions and creates an estimated schedule for transaction execution, estimating when each transaction will be executed. This estimated schedule is used to defer transactions that do not fit in the given execution time window. While the sequencing layer does create an estimated schedule, the actual schedule will likely differ due to dynamic interaction between conflicting transactions, which leads to differences between estimated and actual execution times.

As illustrated in Figure 1, the sequencing layer processes a block[1] of transactions from consensus alongside execution time estimates. While these estimates are derived from transaction properties and can be generated as early as the dissemination phase, the specific estimation mechanisms are outside the scope of this work. Furthermore, we define a fair ordering extension based on dissemination and/or consensus data. Our model does not depend on the underlying protocols used to establish this ordering, assuming that there is a causal order among the transactions received from consensus.

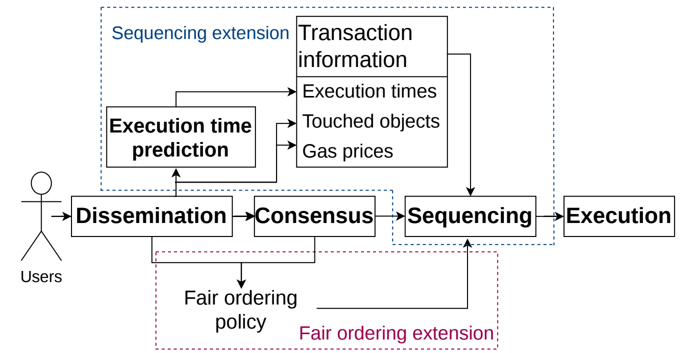


Fig. 1: Modular architecture of a lazy blockchain. The sequencing layer, located in between consensus and execution, is responsible for transaction ordering and congestion control.

A. Heuristic ordering

The current transaction ordering mechanism in Sui employs a linear ordering heuristic based on the gas price for shared objects. This approach builds a schedule in a greedy manner, one transaction at a time. With such an approach, a highly congestive transaction can be sequenced first, which might lead to subsequent executor downtime when the following transactions conflict with it. An illustration of such a sequence is illustrated in Figure 2. In this example, which involves four workers, greedily scheduling transaction 1 (bottom left), which conflicts with transactions 2 to 5, prevents scheduling any of the other transactions and leads to more deferred transactions, possibly suboptimal gas revenue and increased downtime compared to an alternative sequencing (bottom right) that would schedule transactions 2 to 5 and defer transaction 1.

Using another ordering criterion, instead of gas price, in a greedy strategy would possibly lead to similar scenarios where a highly congestive transaction is scheduled first.

[1] Throughout this work we use the term "block" to refer to an atomic unit of transactions that are sequenced and executed together. While we recognize that in DAG-based architectures a single commit can encompass multiple vertices, we use the term "block" to stay protocol-agnostic.

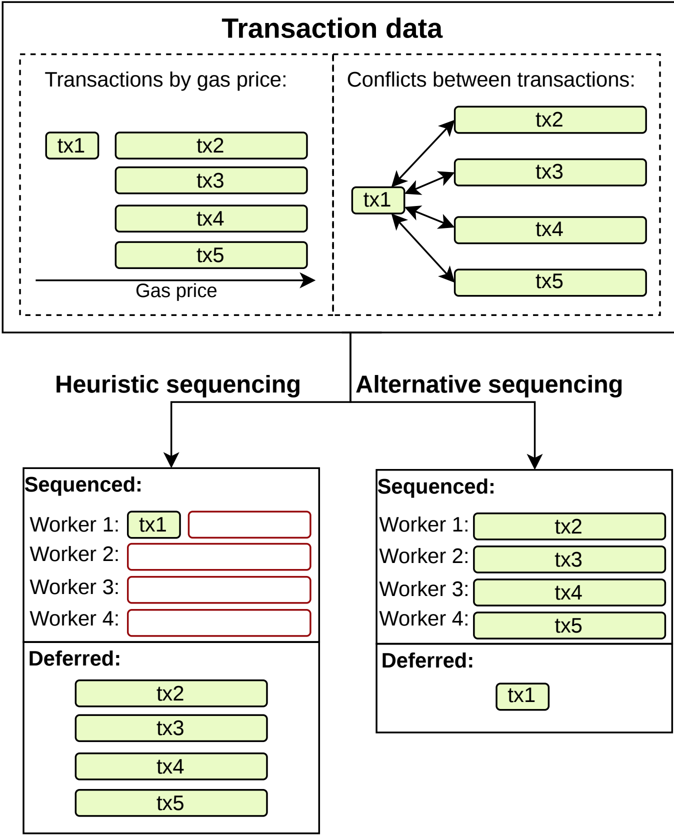


Fig. 2: Greedy sequencing of transactions based on gas price (left) compared to a feasible alternative (right). The size of transactions corresponds to transaction execution time. The greedy algorithm can be sub-optimal and defer many transactions.

Instead, we propose to not only account for estimated execution times and transaction conflicts when estimating the scheduling of transactions, but to also incorporate them into the initial ordering step.

B. Sequencing objectives

We look at sequencing as a combinatorial problem that aims to determine a schedule for transactions that optimizes for a performance metric. For example, if the goal is to schedule all the given transactions in the shortest amount of time, one would minimize the makespan. Alternatively, if the goal is to reach the highest possible throughput, then one would aim to maximize the number of transactions that fit into a given timeframe. In this work, we focus on maximizing the sum of the transaction fees of scheduled transactions. We define the fee of a transaction as its execution time multiplied by its gas price. Optimizing for transaction fees ensures that the less congestive transactions are prioritized, while still allowing the more congestive transactions to be scheduled, as long as their gas price is sufficiently high. This approach also guarantees maximum profit for the validators, which incentivizes further participation in the blockchain.

C. Sequencing requirements

To be practical, the sequencing step should satisfy three main requirements: (i) it should not introduce a performance bottleneck; (ii) it should be adjustable; and (iii) it should be continuous.

The primary goal of sequencing is to improve the performance of execution. Thus, the sequencer must not interfere with execution and must be able to run in parallel with it. We assume execution is distributed across multiple workers that may finish execution of a block at different times. Therefore, an executor's availability time is not fixed, and the sequencer should be able to terminate the optimization process early and produce a good quality sequence at any time.

The need for adjustability is further highlighted by the inherent overhead that sequencing introduces. Since sequencing adds an additional step to the processing of the block, it increases latency. When congestion is low, there is no need to increase the latency, as even a simple sequence would lead to all the transactions being scheduled. An adjustable sequencer would be able to scale the time dedicated to ordering transactions.

Furthermore, sequencing needs to support continuity. In this model, execution does not wait for a block to be fully processed before moving on to the next block. When a worker finishes with all the assigned transactions from block N, it moves immediately onto block $N+1$ without waiting for other workers. So the sequencer needs to account for some workers becoming free before others.

As illustrated in Figure 3, this adjustability and continuity allow for parallelization between different blockchain layers. The diagram shows how the processing of blocks in one layer might affect the processing of blocks in another layer. It highlights how a blockchain could react to an increase in congestion by reducing the time for dissemination of the current block $b3$ and increasing the time for sequencing of that block. This will lead to increased latency for future blocks, while increasing the resources used for sequencing. Furthermore, the figure demonstrates the realization of continuity through overlapping execution of blocks $b2$ and $b3$. The sequencing phase is terminated early to immediately provide the execution layer with a sequenced block, preventing worker idling.

As we try to keep this work independent from concrete implementations of other blockchain layers, we do not design a mechanism for sequencing control between layers. Instead, these requirements manifest in the design choices of our ordering algorithm. It needs to be anytime to satisfy requirements i and ii and it needs to account for continuity to satisfy requirement iii.

D. Scheduling estimation

While this work focuses on optimizing the ordering step of sequencing, a practical evaluation of ordering requires an estimation of the resulting schedule with final performance dependent on the execution timeline.

To bridge this gap, we introduce the concept of a scheduling function, which is a function that estimates the start time

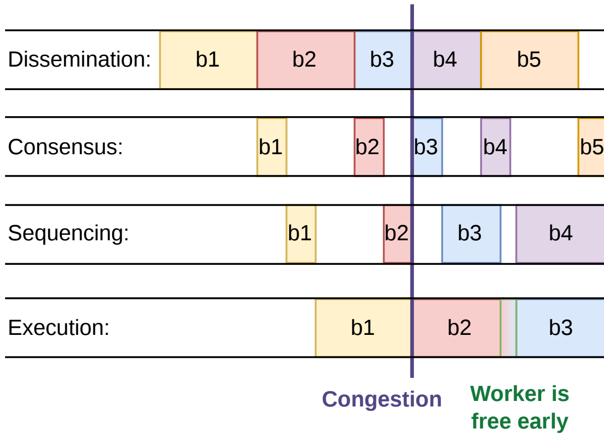


Fig. 3: Processing of blocks $b1$ to $b5$ by the different blockchain layers. Note the increased sequencing time for $b3$, triggered by congestion detected during the execution of $b1$. Furthermore, the sequencing of $b3$ was terminated early to satisfy continuous execution requirement, since a worker becomes free after executing $b2$ while other workers are still executing $b3$.

and assigned worker for a set of transactions when given an ordering of transactions. Although our approach is in principle agnostic to the realization of the scheduling function, in practice and for the purpose of schedule evaluation we provide a concrete scheduling algorithm to serve as a benchmark for evaluating our results.

Since scheduling combinatorial optimization problems are typically NP-hard, we instead rely on a deterministic polynomial-time scheduling algorithm. Our goal is to have a scheduling algorithm that is fast and produces high quality schedules. We rely on a greedy scheduler which schedules the transactions greedily one by one and defers transactions to the next block if they cannot be scheduled at the end of a worker's queue without possibly leading to a conflict with a transaction scheduled on another worker. This is a continuation of the approach proposed by Sui and extended to the case with a fixed number of workers, whereas the original Sui implementation did not assume a concrete number of workers, imagining a system with an unlimited degree of parallelization and focusing only on object availability. The pseudocode of Sui's approach, extended with a restriction for a concrete number of workers, is shown in Algorithm 1.

The introduction of a finite worker pool necessitates a selection strategy to decide which worker will be assigned which transaction. We settled on the Earliest Start Time strategy where transactions are scheduled on the worker with the earliest availability time since it showed the highest throughput during empirical testing.

A straightforward improvement over the aforementioned greedy approach is to not only schedule the transactions at the end of schedules within executors, but to also check if the workers have any gaps in which the transaction can be

Algorithm 1: Greedy scheduler

input : L_{all}, d, m
output : Assignment A, where $A[tx] = (t, w_{id})$ or None

1 $A \leftarrow \emptyset$
2 **for** tx ***in*** L_{all} **do**
3 $\quad WrkFreeAt \leftarrow \emptyset$
4 $\quad$ **for** w ***in*** $0, 1, ..., m-1$ **do**
5 $\quad\quad WrkFreeAt[w] \leftarrow$ find_earliest(A, w, tx)
6 $\quad wrk \leftarrow$ choose_worker($WrkFreeAt$)
7 $\quad$ **if** $wrk == None$ **then**
8 $\quad\quad A[tx] \leftarrow$ None
9 $\quad$ **else**
10 $\quad\quad A[tx] \leftarrow (WrkFreeAt[wrk], wrk)$
11 **return** A

scheduled [27]. This approach is a direct improvement over scheduling transactions after all the previously scheduled ones. The downside of this approach is that it substantially increases the complexity of the algorithm. Before it was sufficient to track only the latest times at which each worker and object are occupied. Now each insertion first has to be compared against a list of gaps within the schedule.

To make the scanning for gaps operation more efficient, we implement this approach using a B-Tree data structure to keep the gaps ordered by their duration and only scan the gaps starting from the smallest gap that could fit the transaction and stopping at the first found gap. This approach is different from the original proposal for gap-filling scheduling. Although it may sometimes produce different schedules, we determined that this implementation has the same average schedule quality as the original implementation, while showing a slight performance improvement with a larger number of transactions. From here on, we consider the greedy gap-filling scheduler as the scheduling function for our problem.

III. System Model and Objectives

Table I summarizes the notation used throughout this paper. The symbols are grouped by the context in which they are introduced: single-shot sequencing and continuous sequencing.

A. Assumptions

We make three important assumptions: (i) the read and write sets of transactions are available; (ii) the transaction execution times are available; and (iii) the system is congested (otherwise there is no need to produce an optimized schedule).

The first assumption is very easy to satisfy. In the pessimistic execution model, the touched object set is known in advance [28, 29]. Each transaction has to explicitly declare its read and write sets. However, even in models where there is no requirement to declare this set, there are static analysis tools that can accurately determine this set [30, 31].

TABLE I: Summary of notation.

Symbol	Description
Single-shot sequencing	
$d \in \mathbb{R}^+$	Execution deadline
$m \in \mathbb{N}^+$	Number of workers
$T_{all} = \{tx_1, \ldots, tx_n\}$	All transactions
$tx_i = (O_i, t_i, g_i)$	Transaction
$O_i = (R_i, W_i)$	Object accesses
$R_i \subseteq \mathcal{I}$	Read set
$W_i \subseteq \mathcal{I}$	Write set
$t_i \in \mathbb{R}^+$	Execution time
$g_i \in \mathbb{R}^+$	Gas price
$L_{all} = [tx_1, \ldots, tx_n]$	Transaction ordering
$\sigma : (L_{all}, d, m) \to \Pi_{L_{all}}$	Scheduling function
$\Pi_{L_{all}} : tx_i \to \text{Either}(\texttt{None}, (s_i, w_i))$	Schedule
$s_i \in \mathbb{R}^+$	Start time
$w_i \in \mathbb{N}^+ \cup \{0\}$	Assigned worker
$T_{sched} \subseteq T_{all}$	Scheduled transactions
$P \in \mathbb{R}^+$	Validator profit
Continuous sequencing	
$d_i \in \mathbb{R}^+$	Execution deadline at block i
$m_i \in \mathbb{N}^+$	Number of workers at block i
$T_{i_{inc}}$	Incoming transactions at block i
$\theta \in \mathbb{N}^+$	Max deferral rounds
$T_{i_{all}}$	All transactions considered at block i
$T_{i_{sched}} \subseteq T_{i_{all}}$	Scheduled transactions at block i
$\delta_i \in \mathbb{R}^+$	Max slack time at block i
$C_i^w \in \mathbb{R}^+$	Latest finish time of worker w at block i
$d_{i_{new}} \in \mathbb{R}^+$	Extended deadline at block i
$O_{tch_j} = R_j \cup W_j$	Touched objects of tx_j
$\alpha_k^i \in \mathbb{R}^+$	Latest scheduled time for object k at block i
$S_i^k \subseteq T_{i_{sched}}$	Transactions touching object k at block i
Fair ordering	
$L_{fair} = [tx_1, \cdots, tx_n]$	Total order over transactions
$G_{fair} = (T_{all}, E)$	Transaction dependency graph

The second assumption is much more difficult to ensure. It is impossible to precisely know the execution path of a transaction and thus its execution time. Even if the transaction is pre-executed, its execution time might depend on its position in the block. Despite this, there has been significant work done on trying to predict the execution time both through historical analysis and simulation, which showed good accuracy [32]. In the experiments we introduce this inaccuracy in predictions and evaluate its impact on the performance of sequencing.

The third assumption is also crucial, as the model aims to maximize the system's utilization of resources. If the incoming transactions cause no congestion, then there is no point in adding another transaction processing step, as any order of transactions would trivially lead to maximum utilization.

B. Single-shot scheduling

Input. Let the execution deadline $d \in \mathbb{R}^+$ be the maximum amount of time a worker can be active for a given block. We consider a system that relies on $m \in \mathbb{N}^+$ workers to execute transactions in parallel. We are given a scheduling function σ that takes as input an ordered list $L_{all} = [tx_1, tx_2, \ldots, tx_n]$ of n transactions and schedules all or part of them on the m workers over a time period d, producing a schedule $\Pi_{L_{all}}$.

A set of all transactions $T_{all} = \{tx_1, \cdots, tx_n\}$ consists of transactions tx_i represented as a tuple (O_i, t_i, g_i). The tuple $O_i = (R_i, W_i)$ of object ids that tx_i accesses consists of a set R_i of object ids that it reads from and a set W_i of object ids that it writes to. Variable $t_i \in \mathbb{R}^+$ is the execution time of tx_i, and $g_i \in \mathbb{R}^+$ is the gas price of a unit of execution time paid by the transaction.

A scheduling function $\sigma : (L_{all}, d, m) \to \Pi_{L_{all}}$ is set in advance and represents a scheduler. It takes L_{all}, d and m as input and produces a schedule $\Pi_{L_{all}} : tx_i \to \text{Either}(\texttt{None}, (s_i, w_i))$. The tuple (s_i, w_i) corresponds to the start time s_i and the worker id w_i on which the transaction has been scheduled. Transactions that are mapped to None are deferred to the following block.

Constraints. Any schedule $\Pi_{L_{all}}$ produced by σ must be subject to several constraints. First, a scheduled transaction tx_i must be scheduled at a positive time ($s_i \geq 0$) and adhere to the deadline ($s_i + t_i \leq d$). It also must be assigned to a valid worker id $w_i \in \{0, 1, \cdots, m-1\}$. Finally, the execution must be conflict-free: if tx_j is scheduled on the same worker as tx_i $(w_i = w_j)$ or if the transactions conflict i.e., $((W_i \cap (R_j \cup W_j)) \neq \emptyset) \vee ((W_j \cap (R_i \cup W_i)) \neq \emptyset)$ then their executions must not overlap ($s_j + t_j \leq s_i \ \vee \ s_i + t_i \leq s_j$).

Objective. Our goal is to find an ordering L_{all} of transactions T_{all}, such that for a given σ we maximize P:

$$P := \sum_{(O_i, t_i, g_i) \in T_{sched}} t_i \cdot g_i$$

where T_{sched} is the set of scheduled transactions:

$$T_{sched} := \{tx_i \in T_{\text{all}} \mid \Pi_{L_{all}}(tx_i) \neq \text{None}\}$$

C. Continuous execution

The above mentioned formulation handles sequencing of transactions one block at a time. But as opposed to consensus, which operates in discrete intervals, transaction execution is a continuous process. There is no need to wait for all the transactions of a previous block to be executed before executing non-conflicting transactions from the next block. To allow for transaction execution continuity, we extend our problem formulation in several ways.

We label parameters based on the block number for which we generate the sequence: $T_{i_{inc}}$, d_i, m_i are the set of transactions, execution deadline and the number of workers at block i. $T_{i_{inc}}$ denotes the set of new transactions that just came in during the new round. The full set of transactions that will be scheduled in round i is denoted $T_{i_{all}}$. We also define θ, which determines the maximum number of rounds for which a transaction can be deferred before it is canceled.

We cancel transactions that have been deferred for too many rounds, and add the rest to the transaction set of the block:

$$T_{i_{all}} = T_{i_{inc}} \cup \{tx_j \mid \Pi_{L_{i-1}}(tx_j) = \text{None} \wedge tx_j \notin T_{(i-\theta)_{inc}}\}$$

We extend the deadline by the difference between the time the earliest worker is free and the deadline at the previous block: let δ_i be the maximum slack time at the end of round i:

$$\delta_i = \max_{w \in [0,1,..m_i-1]} (d_{i_{new}} - C_i^w)$$

where C_i^w is the latest execution end time for worker w at round i:

$$C_i^w = \max\{s_j + t_j \mid \Pi_{L_i}(tx_j) = (s_j, w)\ ,\ tx_j \in T_{i_{sched}}\}$$

We define the extended deadline to be:

$$d_{i_{new}} = d_i + \min(\delta_{i-1}, d_{i-1})$$

We limit the maximum additional execution time per round by the execution time of the previous round, to disallow propagation of multiple rounds.

To ensure that transactions can not be scheduled before the executor is fully done with transactions from a previous block we introduce a constraint: if a transaction tx_j from block i is scheduled at $\Pi_{L_i}(tx_j) = (s_j, w_j)$, then

$$s_j \geq \delta_{i-1} - (d_{i-1} - C_{i-1}^{w_j})$$

Finally, we have to make sure that transactions do not access objects before all the transactions that read from or write to the overlapping objects of the previous block have finished: let $O_{tch_j} = R_j \cup W_j$ for a given $tx_j \in T_i$, then α_k^{i-1} is the latest time at which a transaction touching object k has been scheduled in round $i-1$:

$$\alpha_k^{i-1} = \max_{tx_j \in S_i^k} (s_j + t_j)$$

with S_i^k the set of transactions that touch the object k in round i:

$$S_i^k = \{tx_j \in T_{i_{sched}} \mid k \in O_{tch_j}\}$$

Then a transaction can only be scheduled after the latest time that an object has been freed in the previous round, so if a transaction tx_j from block i is scheduled at $\Pi_{L_i}(tx_j) = (s_j, w_j)$, then:

$$s_j \geq \delta_{i-1} - (d_{i-1} - \max_{k \in O_{tch_j}} (\alpha_k^{i-1}))$$

This approach globally aligns the start times for the new block with the worker that is free the earliest in the previous schedule, while shifting other executors by how late their execution is compared to the earliest executor.

The main benefit of this approach is that it enables the continuity of execution without major changes to the problem structure. The model remains the same, barring the required addition of new hard-coded transactions that represent objects and workers not being free at the start of the round. This allows us to handle sequencing of each block as an independent combinatorial problem and additionally enables varying system parameters, such as d, m or σ, dynamically based on the demand.

D. Fair ordering

Applying the sequencing step requires completely reordering transactions. A strictly opposite approach would be to instead focus on maintaining an order of transactions closely aligned with the order in which they have been submitted to the mempool, called fair ordering [23, 24]. We extend our model to include this constraint in order to measure its impact on the performance.

We make two changes to the model. First, T_{all} is replaced by $L_{fair} = [tx_1, tx_2, \cdots, tx_n]$ which corresponds to the total order over transactions produced by fair ordering. Additionally, the scheduling function must respect the causal order established by this order within L_{fair}, meaning that for every two conflicting transactions tx_i, tx_j where $i < j$: if the transaction positioned later in the order is scheduled, i.e., $\Pi(tx_j) = (s_j, w_j)$, then the earlier transaction must also be scheduled before it, i.e., $\Pi(tx_i) = (s_i, w_i)$ and $s_i + t_i \leq s_j$..

The scheduling algorithm must also be adapted to satisfy the fair ordering constraint. To preserve a simple and greedy design, we assume that the transactions passed to the scheduler are first totally ordered in a way that respects the causal order derived from L_{all}. As a result, the set of candidate schedules available to the scheduler is restricted. Specifically, this set consists of all possible traversals of the dependency graph G_{fair} constructed from L_{fair}.

More specifically, the scheduling algorithm is modified in the following ways. First, if a transaction is deferred, all conflicting transactions that are ordered after the deferred one are also deferred. Second, each transaction can only be scheduled after all conflicting transactions that were ordered before it.

The pseudocode with the highlighted changes is shown in Algorithm 2. Note that the inner workings of the find_earliest function have to be changed to return a time greater than the end time of any previously scheduled conflicting transaction.

IV. Genetic Algorithm for Ordering

If we were to leave out a concrete scheduling function, instead reformulating the goal as determining the scheduling function that leads to an optimal result, the problem would lend itself perfectly to being solved by a constraint solver in the form of a scheduling problem. We opted against this approach as it would be too inefficient, as also found by Chahoki et al. [33], and would diminish the benefit of sequencing by introducing a new bottleneck.

Instead, we focus on transaction ordering under a deterministic scheduling algorithm. This approach allows for the possibility of imperfect information. Sequencing only has access to the execution time estimates. In practice, these values will differ from the predicted ones, and the actual schedule will differ from the predicted one [32].

The most straightforward way of sequencing is to use a heuristic to order transactions. Since the proposed problem has major similarities to a multidimensional extension of the knapsack problem [34], the most natural heuristic is to order the transactions according to their gas price. We also

Algorithm 2: Order-fair greedy scheduler

```
   input  : L_all, d, m
   output : Assignment A, where A[tx] = (t, w_id) or
            None
 1 Def ← ∅
 2 A ← ∅
 3 for tx in L_all do
 4     earliest_start ← latest_conflict_ends(tx, A)
 5     if does_conflict_with_any(tx, Def) then
 6         A[tx] ← None
 7         Def ← Def ∪ {tx}
 8         continue
 9     WrkFreeAt ← ∅
10     for w in 0, 1, ··· , m − 1 do
11         WrkFreeAt[w] ← find_earliest(A, w, tx)
12     wrk ← choose_worker(WrkFreeAt)
13     if wrk == None then
14         A[tx] ← None
15     else
16         A[tx] ← (WrkFreeAt[wrk], wrk)
17         Def ← Def ∪ {tx}
18 return A
```

considered other heuristics: (i) random order; (ii) given order of transactions, which corresponds to the order in which the transactions appear in the historic data, when applicable; and (iii) lowest execution time.

A. Genetic ordering

With a determined scheduling function, the problem lends itself well to being solved through the application of a genetic algorithm. Each candidate solution is encoded as a permutation of the available transactions, where the position of a transaction corresponds to the order in which the scheduler processes them. The application of the scheduling function and the calculation of profit correspond to the fitness function. The high level idea is to start from a population of random sequences and then continuously optimize the population by generating new sequences through crossover and mutation, retaining only the fittest half. This approach is illustrated in Figure 4.

Since in practice the gas price heuristic was found to produce high quality solutions, we initialize our algorithm from a heuristic baseline, rather than from a random sequence. Thus, the goal is to locally refine the initial solution rather than searching for a global optimum.

The pseudocode is shown in Algorithm 3. The techniques used include: seeding, elitism, order crossover (OX) and insertion mutation, which have been shown to be effective for scheduling problems [35].

Seeding. An initial solution is added to the starting population. This solution is made heuristically by ordering transactions based on their gas price [36].

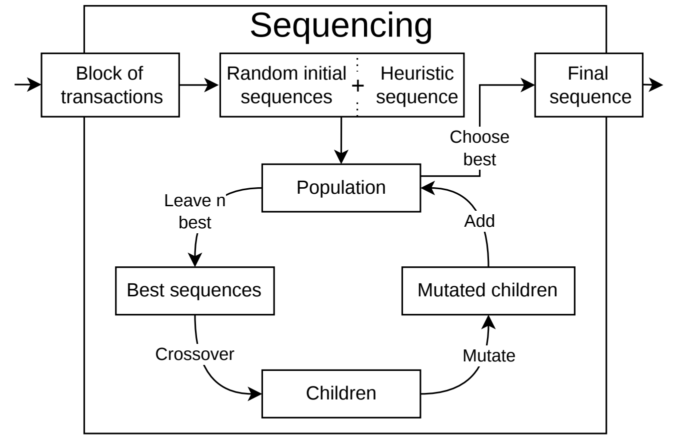


Fig. 4: High level overview of the genetic algorithm for sequencing the transactions.

Algorithm 3: Basis of genetic sequencing

```
   input  : budget, pop_size, T_all, σ, d, m,
   output : L_all
 1 L_init ← highest_gas_price_order(T_all) ; // seeding
 2 P_init ← random_sequences(T_all, pop_size) ∪ {L_init}
 3 P_eval ← eval(P_init, σ, d, m)
 4 budget ← budget − (pop_size + 1)
 5 while budget > 0 do
 6     P_kids ← generate_kids(P_eval, pop_size) ;   // OX
 7     P_kids ← mutate(P_kids) ;              // insertion
 8     for kid ∈ P_kids do
 9         P_eval ← P_eval ∪ {eval(kid, σ, d, m)}
10         budget ← budget − 1
11     P_eval ← order_by_profit(P_eval)
12     P_eval ← limit_population(P_eval, pop_size);
        // elitism
13 return P_eval[0]
```

OX. To generate a child from two parent solution sequences, a slice is taken from one sequence (random location and random length) and is inserted into another sequence at the same spot. Then the resulting sequence is scanned, and if a transaction appears twice, its second appearance is removed [37].

Insertion mutation. To mutate a sequence, a random transaction is removed and inserted into a different position [38].

Elitism. A $(\mu+\lambda)$ survival strategy is employed. This means that each iteration parents (μ) and children (λ) are combined and among them a new population is formed by choosing the sequences with the highest fitness [39].

Seeding and elitism help to ensure faster convergence. To slightly offset this, we found it beneficial to set the mutation probability to a high value above 60 percent. The result is that the genetic algorithm acts similarly to a local search procedure, with a higher degree of exploration within the earlier iterations. Initially, an explicit local search procedure was also tested.

It provided little benefit to the solution quality compared to extending the budget for the genetic algorithm, while requiring a lot of sequence evaluations.

B. Adding fair ordering to the genetic algorithm

When adapting the genetic algorithm to maintain a fair order established by the consensus layer, we have to ensure that the transaction encoding represents a valid total order that respects the causal relations specified by L_{fair}.

We apply the following changes. First, we change the initial seed L_{init} to correspond to the sequence achieved through traversal of G_{fair} by prioritizing transactions with a higher gas price. Then, we generate the rest of the initial parents P_{init} by traversing G_{fair} uniformly at random. We modify the crossover operation so that it iterates over the parents and selects the first transaction whose dependencies are satisfied. The process continues until a valid total order consistent with G_{fair} is generated. We also modify the mutation operator as follows. First, for a randomly selected transaction, determine its maximal neighborhood of non-conflicting transactions within the current total order by identifying the closest conflicting transactions before and after it. Then, relocate the selected transaction to a uniformly at random chosen position within this neighborhood.

V. Performance Evaluation

In the performance evaluation, we do not report the runtime of the algorithms, instead focusing on their performance. The genetic algorithm is an anytime algorithm, so result quality depends on the allotted time. We do not estimate the possible real-world execution time. Instead, we report it in terms of iterations of the estimated sequencer. The runtime of the genetic sequencer is proportional to the number of total evaluations of the estimated sequencer. The number of evaluations is equal to the number of kids per epoch multiplied by the number of epochs. Additionally, the evaluations of children within a single epoch can be done in parallel. To measure the impact of the allotted execution time on performance, we do two runs of the genetic sequencer with the population of 100: for 10 and 50 epochs, corresponding to 1000 and 5000 iterations of the heuristic sequencer, respectively. This allows us to quantify the benefit of additional computation time.

We focus on two scenarios with different solution quality metrics: sustained congestion and a single spike in congestion. In the sustained congestion scenario, we have congestion for the whole duration of sequencing and measure solution quality as the sum of gas prices paid by the scheduled transactions. In the single spike scenario we have congestion for a period of ten blocks and determine the solution quality by how quickly the sequencers managed to clear congestion. We define congestion clearance as the index of the first block after the spike period that contains no deferred transactions.

A. Dataset and parameters

The performance of the proposed sequencing approach is highly dependent on the structure and parameters of the transactions to be scheduled. To obtain measurements that are more representative of the expected real-world performance, we base our datasets on two widely used blockchain systems that support smart contracts: Sui and Ethereum. We took transactions that were executed within a single day on both chains and streamed them to different sequencers. For Ethereum, we took the transactions from block 23359822 up to block 23369821, and for Sui we used transactions from epoch 840. Both represent recent active network conditions and span a sufficient transaction volume to capture realistic congestion dynamics.

In order to do sequencing, we need to extract or generate the following parameters for the transactions:

- Touched objects (O_i) - set of objects/accounts touched by the transactions.
- Execution time (t_i) - amount of computation the transaction takes.
- Gas price (g_i) - price that the party submitting the transaction is paying per execution unit of the transaction.

Other parameters that are independent of transaction data are:

- Transactions per block $(|T_{inc}|)$ - the number of transactions submitted for one block.
- Maximum execution time (d) - the maximum execution time that a single worker can work for per one block.
- Maximum number of deferrals (θ) - the maximum number of blocks for which a transaction can be deferred before it gets canceled.
- Number of executors (m) - the number of executors that can execute the transactions in parallel.

These parameters control how congested the blockchain is. We varied them across experiments to simulate different possible scenarios.

Additionally, all the transactions were initially ordered according to their original positions within the blockchains to better capture temporal fluctuations in gas price trends, demand, and potential conflicts between contemporaneous transactions that may access overlapping sets of objects. To construct a continuous sequence of blocks, we partition the ordered transaction sequence into consecutive groups according to a predefined block size.

We map platform-specific parameters to the general model as summarized in Table II. The execution time was derived from transaction effects for Sui and transaction receipt for Ethereum. The gas price was taken from the transaction data for Sui and from the receipt for Ethereum. Touched objects were determined from changed and unchanged objects specified in transaction effects for Sui and were not collected for Ethereum due to them not being explicitly listed in the available data and requiring high-overhead trace analysis of transactions.

We note that Ethereum uses a base fee system to determine the minimum gas price. Therefore, to better compare transactions across different time periods, we normalize the reported gas used value by this base fee. To do this, we divide the gas

TABLE II: Derivation of model parameters from real-world data.

Parameter	Sui Dataset	Ethereum Dataset
Execution Time	Transaction Effects	Transaction Receipt
Gas Price	Transaction Data	Transaction Receipt
Touched Objects	Shared Objects	N/A (Account-based)

price by the base fee for a given block made during different time periods.

Since the results heavily depend on the data used, we want to first demonstrate the structure of the data and examine the distribution of values, as well as show their correlation with each other. Figure 5 demonstrates the distribution of values for the Sui dataset, and Figure 6 shows the distribution of values for the ETH dataset. From the datasets, we can see that the only two significantly correlated parameters are the number of touched objects and the execution time of transactions ($r = 0.45$), while neither execution time nor number of touched objects is significantly correlated with gas price.

As we mentioned before, we did not use actual data on which accounts were touched by which Ethereum transaction, but we resampled Sui's dataset. To construct the Ethereum-based dataset, we projected Sui's data on object conflicts over Ethereum's execution and gas price records. Specifically, we picked a random point in the Sui dataset and appended this information to the Ethereum stream one transaction at a time, preserving the order in which both datasets were scheduled on the blockchain. This approach introduces a key limitation: it removes the correlation between the number of touched objects and the execution time. For this reason, we consider Ethereum's dataset to be synthetic and refer to it as "Ethereum-based".

As shown in Figure 7, we measured to what degree transactions overlap depending on the order in which they were submitted. The average Jaccard similarity follows a power-law decay as a function of transaction distance, with a sharp initial drop and a heavy tail. This confirms the significance of temporal locality and also shows the presence of several frequently touched global objects.

The prevalence of temporal locality implies that a realistic dataset must consist of locally dependent transactions. It cannot be represented by independent sampling which would lead to a much lower degree of transaction conflicts. Furthermore, the heavy tail indicates the existence of heavily congested shared objects and reaffirms the need for congestion control.

B. Sequencing baselines

We evaluated five different sequencers that rely on heuristics. All of them, except for the one labeled as "Sui", additionally use the gap-filling modification. The sequencers are as follows:

1) Sui - sequencer that orders transactions by descending gas price and does not use gap-filling.
2) Gas Price (GP) - sequencer that orders transactions by descending gas price and does use gap-filling.
3) Lowest Execution Time (LET) - sequencer that orders transactions by ascending execution time.
4) Given - sequencer that orders transactions according to the order in which they have been executed on the blockchain.
5) Random - sequencer that shuffles the order of transactions.

We evaluated our genetic sequencer with two sets of parameters to demonstrate the impact of search depth on solution efficiency.

1) Genetic 100/50 (GE50) - genetic sequencer with population of 100 that runs for 50 epochs.
2) Genetic 100/10 (GE10) - genetic sequencer with population of 100 that runs for 10 epochs.

C. Sustained congestion

For all the following tests, we set the number of executors to 4 and the maximum number of deferrals to 5. The number of transactions was set to 150 and the execution deadline was set to provide around 50-60% inclusion rate for the GP sequencer at 62500 units. The simulation was run for 200 blocks and the results show the cumulative total gas price normalized against the worst-performing (lowest) sequencer at each block. The data is shown from block 20 onward to remove the initial fluctuations and only demonstrate the long-term average result. The results for the Sui data are shown in Figure 8. All of the heuristic approaches except for the GP one show similar performance. The GP sequencer shows a 20% improvement over median of non-GP heuristics and the genetic sequencers exhibit an additional 20% improvement. The GE50 shows a slight improvement (around 5%) over GE10, indicating that, at a population of 100, increasing the number of epochs beyond 10 yields limited gains in efficiency.

For the Ethereum-based dataset, all of the parameters were the same, except for execution time, which was set to 1250000 to provide the same 50-60% inclusion rate. The results for the Ethereum dataset shown in Figure 9, display smaller improvements compared to the Sui dataset. GE10 had only a slight improvement over GP, while GE50 had the same proportional improvement, as it did for Sui. We believe that the main reason for this is the difference in gas price distribution. The gas prices for Sui have low variance as can be seen in Figure 5, while Ethereum's gas prices are much more varied as seen in Figure 6. This means that most of the profit is received by scheduling high-paying transactions. Since the first epochs of the genetic algorithm focus on global search with random initialization of transaction ordering, the algorithm is much less likely to find better sequences. This is because sequencing a high-paying transaction late will have a more severe negative impact on profit. This is also suggested by Sui sequencer having a much better performance in the Ethereum-based dataset than it did in the Sui dataset, since it prioritizes gas price. Due to elitism, the later epochs focus much more on local search, as the transaction set gets filled with similar transactions.

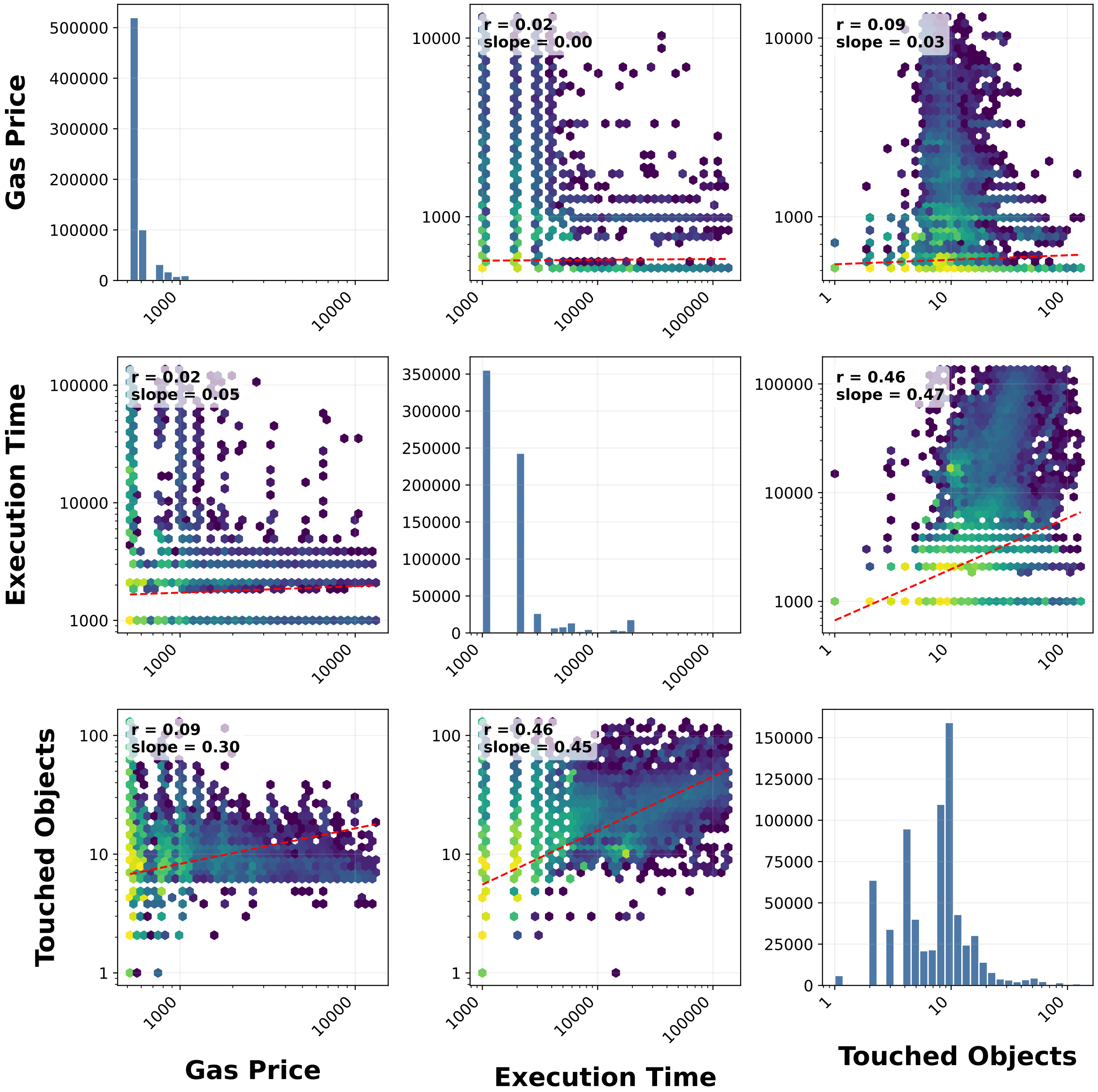


Fig. 5: Log-scaled distribution and density plots for gas price, execution time, and total number of touched objects per transaction for SUI data. The units of these metrics are defined by the specific blockchain. Includes a least-squares regression fit (slope is indicated) and Pearson correlation (r) for each pair of variables. Plots on the diagonal report the counts for each metric, other plots report the value of a metric depending on another one (e.g., number of touched objects depending on gas price).

D. Spike in transactions

To determine how well the proposed genetic sequencing algorithm works in the scenarios where congestion is not sustained, but happens in sporadic bursts, we conducted an experiment where initially each block contains a low amount of transactions, then a high number of transactions for some time after which it goes back to the initial flow of transactions. We kept the number of executors at 4 and the execution deadline at 62500 units for Sui dataset. The maximum number of deferrals was set to 200, so that no transactions would get canceled. The results showing the absolute number of deferred transactions for each block for Sui are shown in Figure 10. Here we see greater variation in the results of heuristic schedulers. The lack of gap filling prevented the Sui sequencer from recovering from the congestion. The Given and GP sequencers achieved similar results, recovering by block 130. LET and Random sequencers recovered around block 90 with LET sequencer showing a more rapid reduction of congestion, but then decreasing more slowly. Both of the genetic sequencers cleared congestion the fastest, with GE50

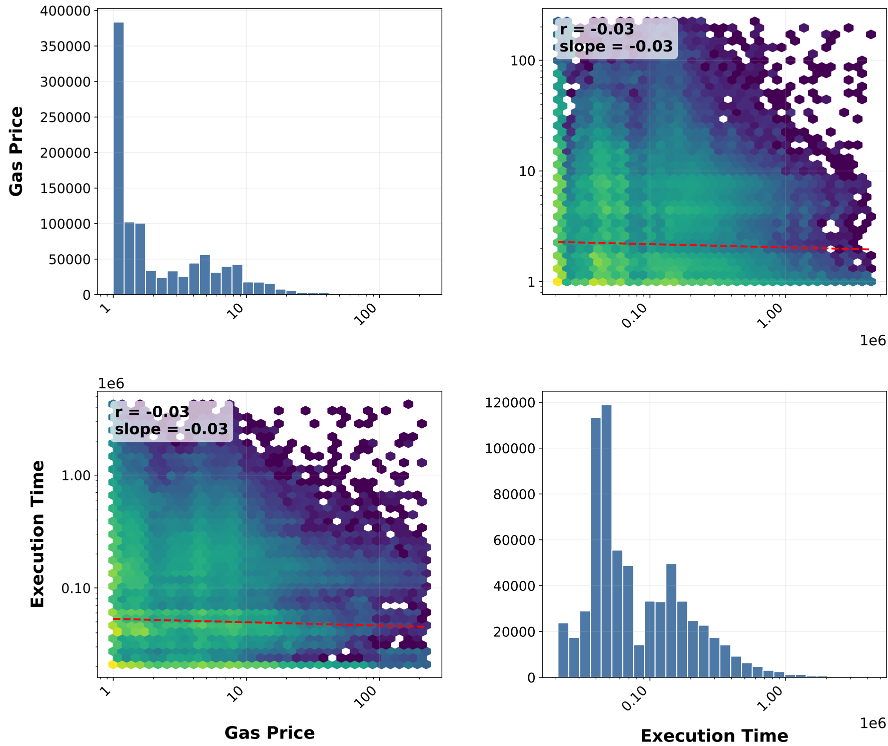


Fig. 6: Log-scaled distribution and density plots for gas price and execution time for Ethereum data. Includes a least-squares regression fit (slope is indicated) and Pearson correlation (r) for each pair of variables.

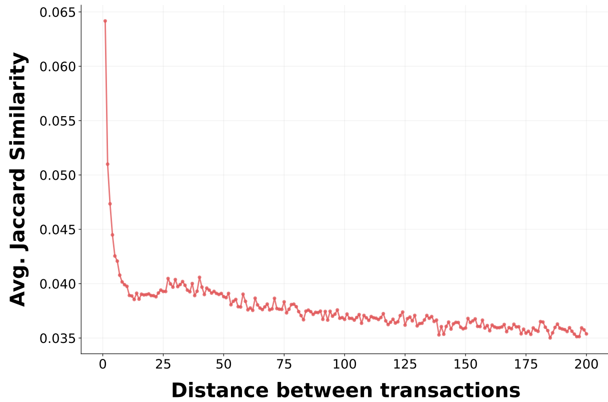


Fig. 7: Average Jaccard similarity (computed over object sets) of transactions based on the distance between them when ordered by execution timestamp. Sample of 50000 random transactions from the full dataset.

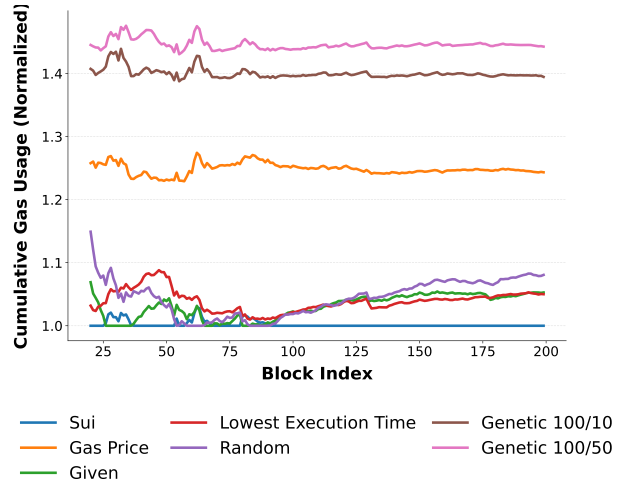


Fig. 8: Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20 for Sui dataset.

doing it by block 60 and GE10 by block 70.

For the Ethereum-based dataset the parameters were the same, with the only difference of execution deadline being, once again, set to 1250000 units. The results are shown in Figure 11. For the Ethereum-based dataset the congestion was less severe and generally cleared much sooner than in the case of Sui. Besides that, the trend of how quickly sequencers manage to resolve congestion mirrors the Sui results with the exception of Random sequencer performing on par with Given and GP sequencers. This is likely because gas prices do not play a role in this scenario.

E. Fair ordering

To determine the impact of fair ordering on performance, we conducted the sustained-congestion experiment under the

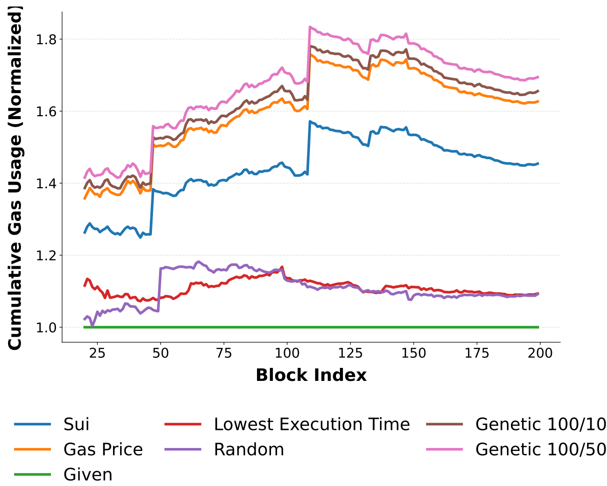


Fig. 9: Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20 for Ethereum dataset.

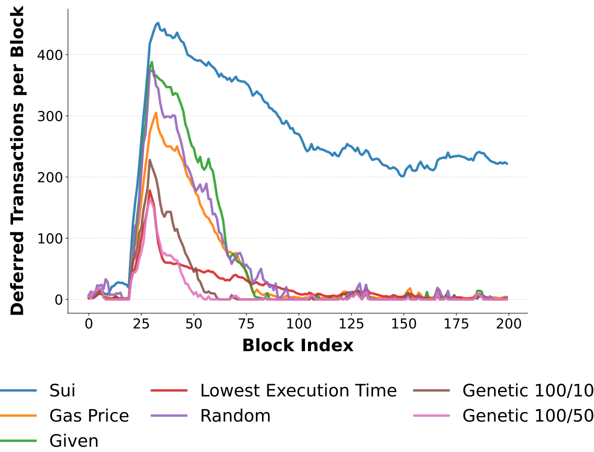


Fig. 10: Total absolute number of transactions deferred under a spike of transactions from block 20 to block 29. With blocks 0-19 having 30 transactions, blocks 20 to 29 having 100 transactions and blocks 30 to 199 having 30 transactions for Sui dataset.

fair ordering constraints. The results for the Sui dataset can be seen in Figure 12 and for the Ethereum-based one in Figure 13. For both cases the fair variants of GP and GE10 produce the same results, suggesting that the fair ordering constraint does not leave sufficient room for reordering causally independent transactions, rendering the genetic algorithm ineffective. Additionally, this experiment highlights that under the sustained congestion, the addition of a fair ordering constraint reduces the profit by 50-60%. We attribute this cost to the high rate of transaction interdependence in the dataset, as can be seen in Figure 7.

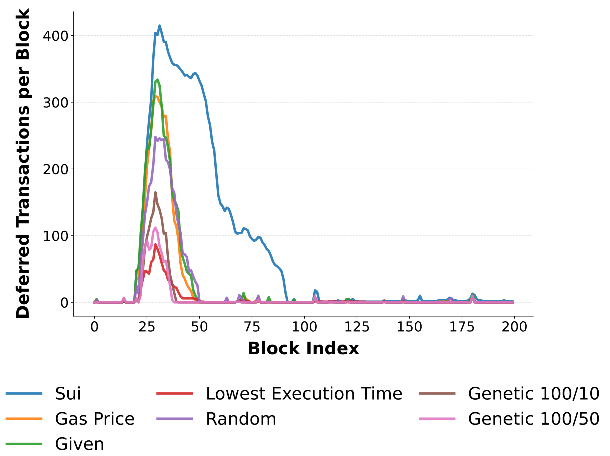


Fig. 11: Total absolute number of transactions deferred under a spike of transactions from block 20 to block 29. With blocks 0-19 having 30 transactions, blocks 20 to 29 having 100 transactions and blocks 30 to 199 having 30 transactions for Ethereum dataset.

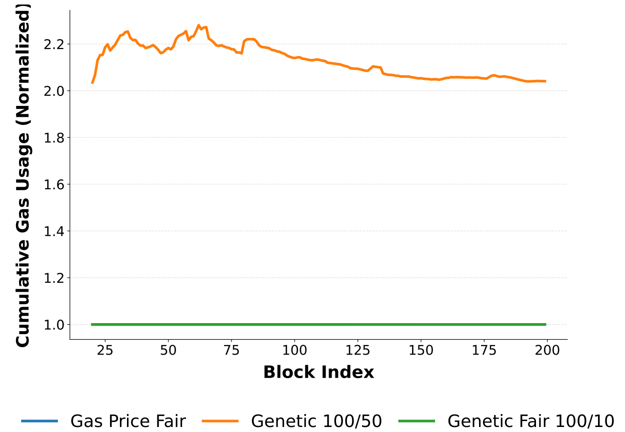


Fig. 12: Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20 for Sui dataset. Sequencers include two sequencers with fair ordering constraints and the GE50, the most efficient non-fair sequencer.

F. Robustness to execution time estimation errors

The most unrealistic assumption that our model hinges on is assuming a perfect prediction of execution time. To check how well the sequencing performs under less accurate prediction of execution time, we repeated the experiments with perturbed transaction execution times. We used the transaction execution times from the data as the predicted execution times for sequencing and when transactions were actually scheduled, we replaced the execution times with their perturbed versions. To generate the perturbed versions of execution times, we applied a multiplicative scaling factor drawn from a log-normal distribution. For each execution time we sampled a normal distribution $X \sim \mathcal{N}(\mu, \sigma^2)$ and computed e^X as a

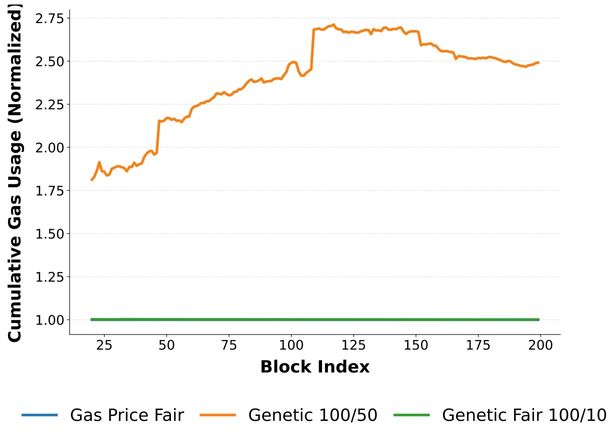


Fig. 13: Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20 for Ethereum dataset. Sequencers include two sequencers with fair ordering constraints, Gas Price Fair and Genetic Fair 100/10, and GE50, the most efficient non-fair sequencer.

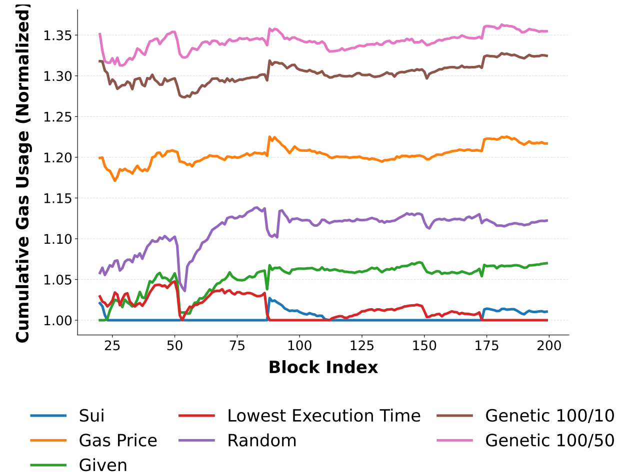


Fig. 14: Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20 for Sui dataset with perturbed transaction execution times

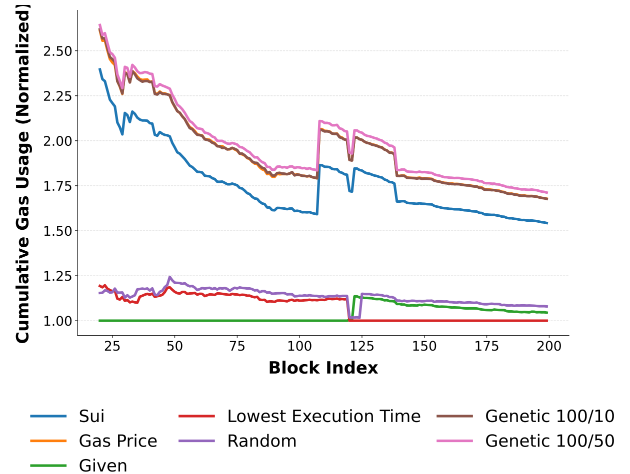


Fig. 15: Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20 for Ethereum dataset with perturbed transaction execution times

scaling factor. We set $\sigma = 0.347$ and $\mu = 0$ to get a 2-sigma range of around $[0.5, 2]$. We multiplied the original execution time by the resulting multiplier. This range was chosen to reflect realistic prediction errors observed in prior work on execution time estimation for blockchain transactions by de Lima Cabral et al. [32] as it provides an average absolute error percentage of around 28%.

Both the sustained congestion and the spike in congestion scenarios were tested with the addition of perturbation for both datasets. The sustained congestion for Sui seen in Figure 14 and congestion spike for Sui and Ethereum seen in Figures 16 and 17 showed no statistically significant difference in speed of congestion clearance compared to the non-perturbed case. The sustained congestion for Ethereum seen in Figure 15 shows that the genetic sequencers performed worse compared to their performance in the analogous experiment without perturbations, with only GE50 showing a very slight improvement over GP.

G. Throughput, latency and full summary

Maximizing throughput and minimizing latency have not been the objectives of this work. Instead of targeting throughput, we decided to focus on maximizing validator revenue. This more closely aligns with the way transactions are currently ordered on most blockchains.

As we do not make any assumptions about the hardware on which sequencing and execution will be run or about the inner workings of the blockchain, we define both throughput and latency as normalized by the time d representing maximum execution time on a single worker. It is equal to 62500 units of execution for the Sui dataset and 1250000 units for the Ethereum-based dataset. Throughput is the average number of transactions executed within this time.

We measure latency as the time it takes from transaction sequencing until it has been executed in the units of d. Since our experiments simulated only the sequencing and execution steps, we assume no additional significant overhead at dissemination and consensus steps and we do not account for the existing latency that they introduce. Additionally, since the simulation has no mechanisms to dynamically vary the time dedicated to sequencing, we assume that in each case sequencing takes time until a worker becomes available.

Latency values below 1 indicate early execution due to worker availability. Values between 1 and 2 indicate normal execution, and values above 2 indicate that the transaction has been deferred. Canceled transactions are unaccounted for,

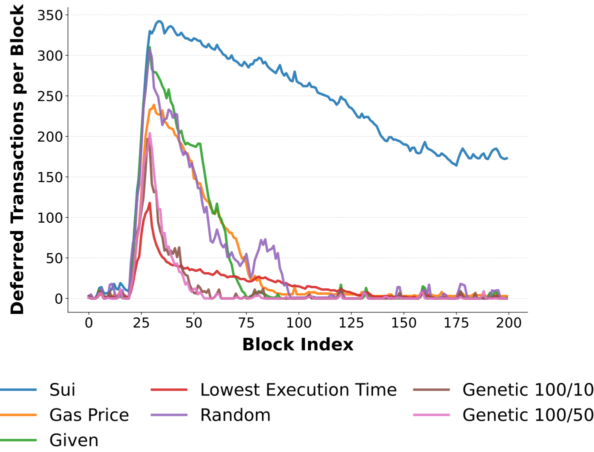


Fig. 16: Total absolute number of transactions deferred under a spike of transactions from block 20 to block 29. With blocks 0-19 having 30 transactions, blocks 20 to 29 having 100 transactions and blocks 30 to 199 having 30 transactions for Sui dataset with perturbed transaction execution times.

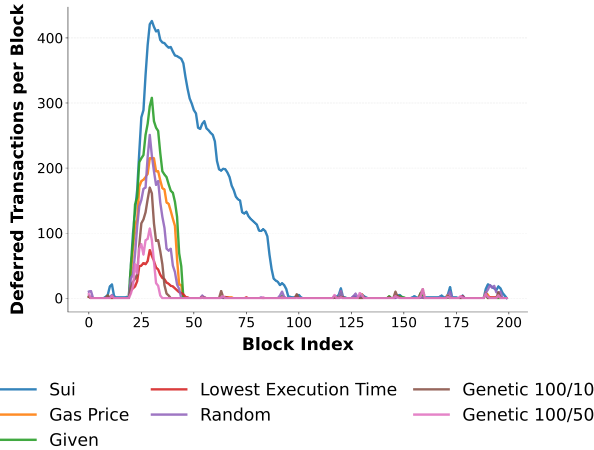


Fig. 17: Total absolute number of transactions deferred under a spike of transactions from block 20 to block 29. With blocks 0-19 having 30 transactions, blocks 20 to 29 having 100 transactions and blocks 30 to 199 having 30 transactions for Ethereum dataset with perturbed transaction execution times.

meaning that the reported latency does not capture the full cost of congestion for affected users.

Results of the experiments are given in Tables III and IV. The genetic sequencer consistently achieves the highest validator profit under sustained congestion. It outperforms the GP baseline by 16% for the Sui dataset and 4% for the Ethereum-based dataset. For the congestion spike scenario, the genetic sequencer clears congestion significantly faster than the baseline, reducing the post-spike recovery time from 60 to 25 blocks on the Sui dataset and from 20 to 8 blocks on the ETH-based dataset, with results being consistent even under perturbation. LET does achieve the highest throughput but at a significant cost to validator profit while the genetic sequencer consistently improves on the baseline throughput. The latency is broadly comparable across non-fair sequencers, though these figures should be interpreted cautiously, as the assumption of equal sequencing time is unrealistic. In practice the genetic algorithm will require more computation time than the heuristic baselines. Fair ordering imposed a profit penalty of approximately 50% and failed to clear congestion on the Sui dataset.

VI. Related Work

To contextualize our contributions, we look at four bodies of related work. First, we examine how transaction sequencing is done in practice on widely used blockchains, ranging from full validator reordering freedom to protocol level congestion control. We then examine the research on fair ordering that disregards congestion control for the sake of transparency and manipulation-resistance. After that, we examine prior efforts to model transaction sequencing as a discrete combinatorial problem. An overview comparing different sequencing approaches is given in Table V. Finally, we briefly overview work that focuses on optimizing transaction execution.

A. Sequencing on popular blockchains

In Bitcoin, transaction sequencing within a block did not play a major role. The main focus of Bitcoin was initially to maintain a consistent total order among blocks of transactions and to ensure that, without controlling the majority of the network's hash rate, it would eventually become impossible to undo previous blocks. The order of transactions within a block was then fully determined by the miner that created it. As Bitcoin transactions could only be of two types, Bitcoin transfers between multiple parties and block rewards [40], the ordering of transactions within the block made no difference. Bitcoin uses the Unspent Transaction Output (UTXO) model, in which each transaction consumes previously created outputs and produces new ones. In this model, since each transaction defines its inputs and outputs independently, the order of transactions within the block does not play a role, only their inclusion. In practice, Bitcoin transactions are included and ordered according to ancestor-feerate sorting, accounting for dependent transactions through Child-Pays-for-Parent (CPFP) [41].

Ethereum employed the same transaction ordering approach as Bitcoin. However, Ethereum uses an account-based model, as opposed to UTXO on Bitcoin. In this model transactions read and write to shared accounts. This means that the output of a transaction is dependent on the state induced by the previous transaction in the same block. This is further amplified by Ethereum's use of smart contracts [21] which can execute arbitrary logic. This means that different orderings of transactions within a block may lead to different final system state. The initial conventional strategy for sequencing transactions within a block was to order them greedily according to their gas price, as employed by the most popular Ethereum client, Geth, which was used by around 76% of the network's validators in 2017 [42, 43]. However, as Daian et al. [22] showed in their

TABLE III: Results of experiments with continued congestion. Throughput is measured as the number of transactions that are scheduled in a block. The normalized amount of gas used represents the perceived revenue, which we aim to maximize. Latency is measured as a multiple of the number of blocks from the start of sequencing to the end of execution.

Sequencer	Throughput ↑	Norm. Gas Used ↑	Latency ↓
Sui			
No Perturbation			
Sui	70.32 ± 1.18	1.41 ± 0.03	1.27 ± 0.02
Gas Price	86.60 ± 1.31	1.76 ± 0.03	1.20 ± 0.02
Given	76.58 ± 1.23	1.49 ± 0.02	1.02 ± 0.02
Lowest Execution Time	**101.58 ± 0.80**	1.49 ± 0.01	**0.90 ± 0.01**
Random	77.29 ± 1.32	1.53 ± 0.02	2.04 ± 0.02
Fair Highest gas price	52.95 ± 0.96	1.00 ± 0.02	2.76 ± 0.10
Genetic 100/10	95.59 ± 1.16	1.97 ± 0.03	1.56 ± 0.02
Genetic 100/50	99.89 ± 1.12	**2.04 ± 0.03**	1.45 ± 0.02
Fair Genetic 100/10	52.96 ± 0.95	1.00 ± 0.02	2.60 ± 0.11
With Perturbations			
Sui	74.51 ± 1.10	1.42 ± 0.04	1.10 ± 0.02
Gas Price	88.80 ± 1.20	1.71 ± 0.04	1.10 ± 0.02
Given	81.69 ± 1.13	1.50 ± 0.03	**0.93 ± 0.02**
Lowest Execution Time	**109.81 ± 0.91**	1.40 ± 0.02	1.10 ± 0.01
Random	83.53 ± 1.24	1.58 ± 0.03	1.83 ± 0.02
Fair Highest gas price	59.53 ± 0.97	1.00 ± 0.01	2.03 ± 0.07
Genetic 100/10	93.36 ± 1.17	1.86 ± 0.04	1.53 ± 0.02
Genetic 100/50	96.69 ± 1.23	**1.90 ± 0.04**	1.54 ± 0.02
Fair Genetic 100/10	59.54 ± 0.97	1.00 ± 0.01	2.03 ± 0.07
ETH-based			
No Perturbation			
Sui	68.99 ± 1.22	2.14 ± 0.15	1.35 ± 0.02
Gas Price	90.08 ± 1.52	2.39 ± 0.16	1.17 ± 0.02
Given	90.89 ± 1.90	1.47 ± 0.07	1.07 ± 0.01
Lowest Execution Time	**125.83 ± 1.07**	1.61 ± 0.07	**1.05 ± 0.01**
Random	89.02 ± 1.66	1.61 ± 0.07	2.17 ± 0.02
Fair Highest gas price	62.67 ± 1.57	1.00 ± 0.05	3.09 ± 0.09
Genetic 100/10	93.50 ± 1.45	2.43 ± 0.15	1.26 ± 0.02
Genetic 100/50	97.47 ± 1.55	**2.49 ± 0.16**	1.23 ± 0.02
Fair Genetic 100/10	62.75 ± 1.57	1.00 ± 0.05	2.99 ± 0.09
With Perturbations			
Sui	74.67 ± 1.22	2.01 ± 0.27	1.28 ± 0.02
Gas Price	96.60 ± 1.55	2.19 ± 0.27	1.10 ± 0.02
Given	96.66 ± 1.69	1.36 ± 0.18	1.08 ± 0.02
Lowest Execution Time	**129.37 ± 0.90**	1.31 ± 0.13	**1.03 ± 0.01**
Random	96.56 ± 1.53	1.41 ± 0.16	2.03 ± 0.02
Fair Highest gas price	68.92 ± 1.45	1.00 ± 0.16	2.79 ± 0.08
Genetic 100/10	96.95 ± 1.45	2.19 ± 0.27	1.27 ± 0.02
Genetic 100/50	100.05 ± 1.49	**2.23 ± 0.27**	1.28 ± 0.02
Fair Genetic 100/10	68.95 ± 1.45	1.00 ± 0.16	2.78 ± 0.07

TABLE IV: Results of experiments with a spike in the number of transactions. Throughput is measured as the number of transactions that are scheduled in a block. Number of blocks to recover from congestion represents the count of consecutive blocks after congestion ends until a block is produced with no deferred transactions. Latency is measured as a multiple of the number of blocks from the start of sequencing to the end of execution.

Sequencer	Throughput ↑	# Blocks to recover from congestion ↓	Latency ↓
Sui			
No Perturbation			
Sui	32.39 ± 0.57	Did not recover	3.82 ± 0.30
Gas Price	33.48 ± 0.74	60	2.20 ± 0.19
Given	33.49 ± 0.69	57	2.38 ± 0.27
Lowest Execution Time	33.49 ± 0.84	162	1.34 ± 0.08
Random	33.49 ± 0.76	66	2.52 ± 0.18
Fair Highest gas price	27.95 ± 0.74	Did not recover	21.68 ± 0.84
Genetic 100/10	**33.50 ± 0.80**	32	1.40 ± 0.09
Genetic 100/50	**33.50 ± 0.88**	**25**	**1.19 ± 0.07**
Fair Genetic 100/10	27.95 ± 0.74	Did not recover	21.26 ± 0.86
With Perturbations			
Sui	32.63 ± 0.65	Did not recover	3.94 ± 0.27
Gas Price	33.48 ± 0.76	59	2.13 ± 0.18
Given	33.49 ± 0.70	62	2.09 ± 0.21
Lowest Execution Time	**33.50 ± 0.91**	126	**1.23 ± 0.06**
Random	**33.50 ± 0.81**	88	2.32 ± 0.15
Fair Highest gas price	26.64 ± 0.78	Did not recover	28.16 ± 1.17
Genetic 100/10	**33.50 ± 0.91**	28	1.29 ± 0.06
Genetic 100/50	**33.50 ± 0.84**	**25**	1.24 ± 0.06
Fair Genetic 100/10	26.64 ± 0.78	Did not recover	28.08 ± 1.18
ETH-based			
No Perturbation			
Sui	33.49 ± 0.64	62	2.80 ± 0.31
Gas Price	**33.50 ± 0.80**	20	1.28 ± 0.11
Given	**33.50 ± 0.79**	20	1.35 ± 0.12
Lowest Execution Time	**33.50 ± 0.96**	23	**0.86 ± 0.04**
Random	**33.50 ± 0.83**	21	1.35 ± 0.10
Fair Highest gas price	**33.50 ± 0.84**	81	3.64 ± 0.33
Genetic 100/10	**33.50 ± 0.89**	10	1.00 ± 0.06
Genetic 100/50	**33.50 ± 0.97**	**8**	0.98 ± 0.04
Fair Genetic 100/10	**33.50 ± 0.84**	81	3.49 ± 0.33
With Perturbations			
Sui	**33.50 ± 0.62**	72	3.14 ± 0.34
Gas Price	**33.50 ± 0.88**	15	1.19 ± 0.11
Given	**33.50 ± 0.84**	15	1.27 ± 0.13
Lowest Execution Time	**33.50 ± 0.97**	18	**0.89 ± 0.03**
Random	**33.50 ± 0.83**	14	1.21 ± 0.07
Fair Highest gas price	33.39 ± 0.96	82	4.96 ± 0.38
Genetic 100/10	**33.50 ± 0.91**	9	1.01 ± 0.05
Genetic 100/50	**33.50 ± 0.99**	**5**	0.95 ± 0.03
Fair Genetic 100/10	33.39 ± 0.96	82	4.98 ± 0.38

work, strategic rearrangement of transactions by a validator and the inclusion of its own transactions could produce significant additional validator profit. Such phenomenon was called Miner Extractable Value (MEV). After this issue became widely known, an entire system of blockspace actions, called MEV-boosting, appeared. Transaction order within blocks was now optimized by separate builder actors in exchange for a share of the extracted value. MEV-boosting has rapidly gained popularity, with around 90% of blocks being created through MEV-boost by the end of 2022 [44]. The rapid adoption is explained by the fact that MEV revenue aligns closely with validator interests, increasing the median validator block reward by approximately 250400%. [45, 46] and thus serving as a further incentive to the builders for participation. Nevertheless, the adoption of such a system has significantly undermined transparency and decentralization. MEV was often extracted by submitting transactions privately to builders or by specifying inclusion requirements called search bundles, such as positioning of transactions [47].

Sui is a DAG-based blockchain that differentiates between owned and shared objects, with owned objects using UTXO-like ordering and shared objects using account-like ordering. Transactions touching only owned objects avoid the overhead of total ordering, only requiring partial order established by their owner and confirmed by validators. There is still a requirement to agree on which transaction to accept if there are conflicting transactions signed by the owner, which can be done through certification or consensus. Transactions touching shared objects still require global sequencing relative to one another [48]. Sui's sequencing is also responsible for congestion control. The initial sequencing of transactions is based exclusively on gas price. After that the estimated execution time and information about touched objects are used to determine which transactions to defer to a later commit. The sequencer keeps track of the queue per object and processes transactions one by one. Transactions that do not fit within the execution budget of objects are deferred [25, 26].

Solana is another popular blockchain that also emphasizes high throughput [49]. Its sequencer went through multiple major changes over time. Initially, the sequencer had multiple independent transaction queues on each of the threads. Within one thread the transactions were ordered by gas price, but independence of threads meant that there was no global order, and transactions were sequenced non-deterministically. A later version of the sequencer (update v1.18) first put the transactions into a max-heap and only then distributed them between threads. However, the Solana protocol does not specify how sequencing must occur. This led to the emergence of an alternative Solana validator client Jito that uses MEV-boost [50]. This illustrates that when transaction sequencing is not specified at the protocol level, economic incentives can often lead validators toward opaque, profit-maximizing sequencing techniques.

B. Fair ordering

The lack of transaction ordering transparency and the use of MEV-boosting have motivated research into fair transaction ordering [51]. The main goal of fair transaction ordering is to ensure that the transaction order aligns as closely as possible with the real-world time at which the transactions have been submitted. There are currently two primary approaches to establishing a fair order: ordering linearizability and batch order fairness. In ordering linearizability, first defined in Pompe [24], validators submit timestamps corresponding to when each transaction was received, and transactions are then ordered based on the median of collected timestamps. In batch order fairness systems, such as Themis [23], each validator submits an ordering of transactions. The final ordering is then achieved by combining validators' individual orderings.

Fair ordering ensures fairness at the protocol level and helps mitigate MEV-boosting. However, it generates a very rigid transaction order, provides no mechanism for congestion control, and does not take into account validator incentives.

C. Combinatorial optimization for sequencing

Under a deterministic scheduling algorithm, transaction sequencing is equivalent to determining block building. If all transactions must be executed sequentially and if we assume that the execution time of transactions is independent of their position in the sequence, the problem is equivalent to the knapsack problem, meaning that it is NP-hard [34].

A concurrent and independently motivated work by Karmegam et al. [52] takes a similar approach to our work by modeling block construction as a mixed-integer program with validator profit as the objective. In contrast to our work, they focus on the scheduling layer and assume a predefined dependency set, and implicitly enforce an ordering constraint analogous to our fair ordering extension. They also propose a sophisticated heuristic for transaction scheduling, which performs the same as a simple greedy gas price heuristic. This mirrors our own finding that the genetic algorithm provides no improvement over the gas-price baseline when fair ordering is enforced.

Another work that has major parallels to ours is Conthereum by Chahoki et al. [33], though it diverges in several critical aspects. In both Conthereum and our work, transaction sequencing is modeled in a parallel execution setting as a job shop scheduling problem with full freedom to reorder transactions. In their analysis they require that all transactions be scheduled and optimize for the makespan within a single block.

Our work further differs from both Karmegam et al. and Chahoki et al. in two respects. First, we extend the problem to a continuous execution setting where execution of successive blocks can overlap. Second, in our experiments we account for perturbations between expected and actual execution time by applying realistic execution time perturbations.

TABLE V: Comparison of different transaction sequencing approaches. Gives an overview of the criteria used for transaction ordering and evaluates to which extent the resulting orderings are deterministic, are resistant to manipulation, specifically preventing validators from arbitrarily reordering transactions to their own advantage, and are aligned with validator incentives.

System	Ordering Criteria	Determinism	Manipulation Resistance	Validator Alignment
Bitcoin	CPFP	High	Moderate	High
Eth (pre-MEV)	Gas price	High	Low	Moderate
Eth (MEV-Boost)	Builder bid	Low	Low	Very High
Sui	Gas price	High	High	Moderate
Solana (<v1.18)	Priority fee (per-thread)	Low	Low	Low
Solana (>v1.18)	Priority fee (global heap) / Builder bid	Moderate	Moderate	High
Fair Ordering	Arrival timestamps	Very High	Very High	Low
Proposed and Conthereum	Gas price + execution time + touched objects	High	High	High

D. Execution

Most of the works in this area focus on improving the execution layer. Pilotfish [13] is a distributed execution engine that consists of a cluster of executors where nodes use the given transaction order and parallelize execution of consecutive independent transactions by dynamically assigning them to workers. It focuses on generalizing the transaction execution in a crash-recovery setting while minimizing the message overhead. HTFabric and Fabric++ [53, 54], which are extensions of Hyperledger Fabric, go further by reordering transactions during execution to increase parallelism. In practice, their approach prioritizes independent transactions. Although this improves throughput, it creates a bias against more interdependent transactions.

While these works have primarily relied on explicitly declared sets of touched objects, other work has parallelized execution without requiring such declarations in advancey. Anjana et al. [55] apply software transactional memory (STM) to maximize parallelism at the cost of redundant computation by speculatively executing transactions in parallel and rolling back in case of conflicts. Block-STM [19] refines this by updating the estimated dependency sets after each rollback, reducing unnecessary re-executions.

VII. Discussion

The performance of the sequencing step was evaluated under existing transaction data. This means that the submitted transactions did not account for the degree of congestion that they might cause when specifying a gas price, as evidenced by the lack of correlation between gas price and number of touched objects as seen in Figures 5 and 6. With the introduction of this step, transactions may start to account for this and adjust their gas prices accordingly. Moreover, in this work we considered only a static gas fee as specified by the user submitting the transaction. Recent work has explored multi-dimensional and dynamic pricing mechanisms better suited for parallel execution environments [56, 57]. Further research into how our sequencing approach interacts with such pricing models would be valuable.

In this work, we relied on the definition of fair ordering that first imposes a total order on the transactions, from which we derived a causal order. Another approach for achieving total order is batch order fairness [23]. This approach might lead to the creation of dependency cycles and would allow us to choose how they are resolved. This might lead to better performance if there are many cycles, but in practice, under realistic assumptions where the majority of nodes try their best to order transactions according to their real-world timestamps, such cycles are exceedingly rare and are unlikely to lead to a major improvement.

In this work, we propose a genetic algorithm for sequencing transactions. This represents one approach; there are other methods, such as GRASP or simulated annealing. The primary objective of this work was to demonstrate the value of the sequencing step. Our results show that even a straightforward approach can lead to a significant improvement.

VIII. Conclusion

In this work, we have formalized a blockchain-independent model for evaluating the impact of transaction ordering on validator profit and congestion clearance speed. We then extended this model to account for continuity of execution. We proposed a genetic ordering algorithm that takes into account transaction execution time estimates and object conflict information. Our results demonstrate that this approach outperforms the heuristic baseline, increasing validator profit by approximately 15%, and improves congestion control in a high congestion scenario. We have also shown that the performance of this approach does not degrade significantly under realistic execution time estimation errors for the purpose

of congestion control. Performance also does not degrade significantly in terms of profit maximization when gas prices exhibit little variation.

We also integrated a fair ordering constraint and found that under such an assumption, sequencing provides no benefit. Moreover, we found that by introducing fair ordering, validators lose around 50 to 60% of their profit under sustained congestion.

References

[1] W. Li and M. He. "Comparative Analysis of Bitcoin, Ethereum, and Libra". In: *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*. 2020, pp. 545–550.

[2] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer. "On Scaling Decentralized Blockchains". In: *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 106–125.

[3] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. "Narwhal and tusk: a dag-based mempool and efficient bft consensus". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 34–50.

[4] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 1187–1201.

[5] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias. "Bullshark: Dag bft protocols made practical". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022, pp. 2705–2718.

[6] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen. "Performance Analysis of Consensus Algorithm in Private Blockchain". In: *2018 IEEE Intelligent Vehicles Symposium (IV)*. 2018, pp. 280–285.

[7] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. "All you need is dag". In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 2021, pp. 165–175.

[8] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 34–50. URL: https://doi.org/10.1145/3492321.3519594.

[9] K. Babel, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, A. Koshy, A. Sonnino, and M. Tian. "Mysticeti: Reaching the Latency Limits with Uncertified DAGs." In: *NDSS*. 2025.

[10] N. Polyanskii, S. Müller, and I. Vorobyev. "Starfish: A high throughput BFT protocol on uncertified DAG with linear amortized communication complexity". In: *IACR Cryptol. ePrint Arch.* 2025 (2025), p. 567. URL: https://api.semanticscholar.org/CorpusID:277637515.

[11] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber. "Optimized execution of business processes on blockchain". In: *Business Process Management: 15th International Conference, BPM 2017, Barcelona, Spain, September 10–15, 2017, Proceedings 15*. Springer. 2017, pp. 130–146.

[12] E. Albert, P. Gordillo, A. Hernández-Cerezo, A. Rubio, and M. A. Schett. "Super-optimization of smart contracts". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.4 (2022), pp. 1–29.

[13] Q. Kniep, L. Kokoris-Kogias, A. Sonnino, I. Zablotchi, and N. Zhang. *Pilotfish: Distributed Execution for Scalable Blockchains*. 2025. arXiv: 2401.16292 [cs.DC].

[14] M. J. Amiri, D. Agrawal, and A. E. Abbadi. "ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems". In: *CoRR* abs/1902.01457 (2019). arXiv: 1902.01457.

[15] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi. "A Transactional Perspective on Execute-order-validate Blockchains". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 543–557. URL: https://doi.org/10.1145/3318464.3389693.

[16] C. Cachin et al. "Architecture of the hyperledger blockchain fabric". In: *Workshop on distributed cryptocurrencies and consensus ledgers*. Vol. 310. 4. Chicago, IL. 2016, pp. 1–4.

[17] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen. "Adding Concurrency to Smart Contracts". In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. PODC '17. Washington, DC, USA: Association for Computing Machinery, 2017, pp. 303–312. URL: https://doi.org/10.1145/3087801.3087835.

[18] S. Das, V. Krishnan, and L. Ren. *Efficient Cross-Shard Transaction Execution in Sharded Blockchains*. 2021. arXiv: 2007.14521 [cs.CR].

[19] 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: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 232–244.

[20] V. Saraph and M. Herlihy. *An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts*. 2019. arXiv: 1901.01376 [cs.DC].

[21] V. Buterin. *Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform*. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[22] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability". In: *2020 IEEE symposium on security and privacy (SP)*. IEEE. 2020, pp. 910–927.

[23] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan. "Themis: Fast, strong order-fairness in byzantine consensus". In: *Proceedings of the 2023 acm sigsac conference on computer and communications security*. 2023, pp. 475–489.

[24] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi. "Byzantine ordered consensus without byzantine oligarchy". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 633–649.

[25] S. Foundation. *Streamlining Transactions with Sui's Shared Object Congestion Control*. Sui Blog. Sept. 2024. URL: https://blog.sui.io/shared-object-congestion-control/.

[26] S. Foundation. *Object-Based Local Fee Markets*. Sui Documentation. 2024. URL: https://docs.sui.io/guides/developer/objects/local-fee-markets.

[27] C. U. Ileri, A. Cullen, O. Saa, R. Overko, and L. Vigneri. "Enhanced Transaction Sequencing for Modular Distributed Ledgers". In: *2025 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE. 2025, pp. 1–3.

[28] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino, B. Williams, and L. Zhang. *Sui Lutris: A Blockchain Combining Broadcast and Consensus*. 2024. arXiv: 2310.18042 [cs.DC].

[29] S. Foundation. *Sealevel - Parallel Processing Thousands of Smart Contracts*. Sept. 2019.

[30] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. "Smartcheck: Static analysis of ethereum smart contracts". In: *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*. 2018, pp. 9–16.

[31] A. Z. Chahoki and M. Roveri. "Static analysis for detecting transaction conflicts in ethereum smart contracts". In: *arXiv preprint arXiv:2507.04357* (2025). arXiv: 2507.04357 [cs.DC].

[32] D. R. de Lima Cabral, P. Antonino, and A. C. A. Sampaio. "Demystification and near-perfect estimation of minimum gas limit and gas used for Ethereum smart contracts". In: *Journal of Cloud Computing* 14.1 (2025), p. 29.

[33] A. Z. Chahoki, M. Herlihy, and M. Roveri. "Contheureum: Concurrent ethereum optimized transaction scheduling for multi-core execution". In: *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE. 2025, pp. 1–10.

[34] S. Dos Santos, C. Chukwuocha, S. Kamali, and R. K. Thulasiram. "An Efficient Miner Strategy for Selecting Cryptocurrency Transactions". In: *2019 IEEE International Conference on Blockchain (Blockchain)*. 2019, pp. 116–123.

[35] R. Cheng, M. Gen, and Y. Tsujimura. "A tutorial survey of job-shop scheduling problems using genetic algorithmsI. representation". In: *Computers Industrial Engineering* 30.4 (1996), pp. 983–997. URL: https://www.sciencedirect.com/science/article/pii/0360835296000472.

[36] B. A. Julstrom. "Seeding the population: improved performance in a genetic algorithm for the rectilinear Steiner problem". In: *Proceedings of the 1994 ACM Symposium on Applied Computing*. SAC '94. Phoenix, Arizona, USA: Association for Computing Machinery, 1994, pp. 222–226. URL: https://doi.org/10.1145/326619.326728.

[37] L. Davis et al. "Applying adaptive algorithms to epistatic domains." In: *IJCAI*. Vol. 85. 1985, pp. 162–164.

[38] K. Jebari, M. Madiafi, et al. "Selection methods for genetic algorithms". In: *International Journal of Emerging Sciences* 3.4 (2013), pp. 333–344.

[39] K. A. D. Jong. "An Analysis Of The Behavior Of A Class Of Genetic Adaptive Systems". In: 1975. URL: https://api.semanticscholar.org/CorpusID:57626488.

[40] S. Nakamoto. *Bitcoin whitepaper*. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[41] J. Messias, M. Alzayat, B. Chandrasekaran, K. P. Gummadi, P. Loiseau, and A. Mislove. "Selfish & opaque transaction ordering in the Bitcoin blockchain: the case for chain neutrality". In: *Proceedings of the 21st ACM Internet Measurement Conference*. 2021, pp. 320–335.

[42] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey. "Measuring ethereum network peers". In: *Proceedings of the Internet Measurement Conference 2018*. 2018, pp. 91–104.

[43] go-ethereum contributors. *go-ethereum miner/worker.go (commit 290e851)*. 2017. URL: https://github.com/ethereum/go-ethereum/blob/290e851f57f5d27a1d5f0f7ad784c836e017c337/miner/worker.go.

[44] B. Öz, D. Sui, T. Thiery, and F. Matthes. "Who wins ethereum block building auctions and why?" In: (2024). arXiv: 2407.13931 [cs.DC].

[45] Nero.eth. *Is it worth using MEV-Boost?* June 2024. URL: https://ethresear.ch/t/is-it-worth-using-mev-boost/19753.

[46] D. Mancino, A. Leporati, M. Viviani, and G. Denaro. "A Role and Reward Analysis in Off-Chain Mechanisms for Executing MEV Strategies in Ethereum Proof-of-Stake". In: *Distrib. Ledger Technol.* 4.3 (Aug. 2025). URL: https://doi.org/10.1145/3672405.

[47] Flashbots. *Understanding Bundles*. 2025. URL: https://docs.flashbots.net/flashbots-mev-share/searchers/understanding-bundles.

[48] R. Overko. "A Study on Shared Objects in Sui Smart Contracts". In: *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2024, pp. 1–7.

[49] A. Yakovenko. *Solana: A new architecture for a high performance blockchain v0. 8.13*. 2018. URL: https://solana.com/solana-whitepaper.pdf.

[50] Lostin. *The Truth about Solana Local Fee Markets*. Jan. 2025. URL: https://www.helius.dev/blog/solana-local-fee-markets.

[51] M. Kelkar, S. Deb, and S. Kannan. "Order-fair consensus in the permissionless setting". In: *Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop*. 2022, pp. 3–14.

[52] A. Karmegam, L. Kiffer, and A. F. Anta. *Exploiting Multi-Core Parallelism in Blockchain Validation and Construction*. 2026. arXiv: 2602.03444 [cs.DC].

[53] J. Song, J. Jeong, J. Lee, I. Na, and M.-S. Kim. "HTFabric: A Fast Re-ordering and Parallel Re-execution Method for a High-Throughput Blockchain". In: *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*. CIKM '24. Boise, ID, USA: Association for Computing Machinery, 2024, pp. 2118–2127. URL: https://doi.org/10.1145/3627673.3679606.

[54] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. "Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric". In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, pp. 105–122. URL: https://doi.org/10.1145/3299869.3319883.

[55] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani. *An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts*. 2019. arXiv: 1809.01326 [cs.DC].

[56] B. Acilan, A. Constantinescu, L. Heimbach, and R. Wattenhofer. "Transaction fee market design for parallel execution". In: (2025). arXiv: 2502.11964 [cs.DC].

[57] S. Wadhwa, A. Yaish, F. Zhang, and K. Nayak. *Perils of Parallelism: Transaction Fee Mechanisms under Execution Uncertainty*. 2026. URL: https://eprint.iacr.org/2026/649.