

The Execution Dilemma in Pessimistic Blockchains: Profitability XOR Fair Ordering

Artjom Pugatsov  

Delft University of Technology, The Netherlands

Can Umut Ileri  

IOTA Foundation, Berlin, Germany

Delft University of Technology, The Netherlands

Jérémie Decouchant  

Delft University of Technology, The Netherlands

Abstract

The successive generations of consensus algorithms progressively displaced the performance bottleneck of blockchains to the execution layer. Recent works address the execution performance bottleneck by parallelizing the execution of non-conflicting transactions. Historically, execution closely followed the consensus-level transaction ordering determined by validators, a practice highly susceptible to Maximal Extractable Value (MEV) exploitation. Conversely, recent academic proposals introduce rigid fair-ordering protocols that are bound to severely restrict transaction reordering at the execution layer. Parallel execution frameworks optimize the sequencing layer, which lies between consensus and execution and assembles transaction batches from committed consensus blocks and transmits them to execution workers to maximize both execution parallelism and realized transaction fees. To achieve this optimization, current sequencing implementations may defer transactions. Importantly, these implementations do not maintain order-fairness properties. To the best of our knowledge, preserving these properties currently requires sequential execution, which would drastically reduce both profitability and performance.

In this work, we address the tension between validator profit and order fairness using a dynamic optimization framework. We introduce a blockchain-agnostic model for transaction sequencing in a continuous setting where block sequencing and execution run concurrently. Consequently, when sequencing cannot be completed within the available time window, our framework dynamically returns its best intermediate result. Within this framework, we propose an anytime genetic algorithm that utilizes gas prices, object sets, and predicted execution times to optimize schedules. We also augment this algorithm to optionally maintain fair-ordering. We evaluate our approach with real-world datasets from Sui and Ethereum, and demonstrate that it increases validator profit by approximately 15% and accelerates congestion relief speed by up to 58%. Furthermore, we quantify the impact of fair-ordering constraints, showing that they can reduce validator profit by 50% to 60% during periods of high congestion. We provide the first evidence that enforcing strict fair ordering might effectively nullify the advantages of advanced sequencing.

2012 ACM Subject Classification Theory of computation → Online algorithms; Applied computing → Electronic commerce

Keywords and phrases Blockchain, Execution layer, Sequencing, Byzantine Fault Tolerance

Digital Object Identifier 10.4230/LIPIcs.AFT.2026.27

Related Version *Full Version*: <https://arxiv.org/abs/2604.23266> [50]

Supplementary Material

Software (Source Code): <https://github.com/Artjom-Pugatsov/genetic-sequencing>



© Artjom Pugatsov, Can Umut Ileri, and Jérémie Decouchant;
licensed under Creative Commons License CC-BY 4.0

8th Conference on Advances in Financial Technologies (AFT 2026).

Editors: Aggelos Kiayias and Maria Kyropoulou; Article No. 27; pp. 27:1–27:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Historically, blockchain systems have faced significant performance limitations. For example, Bitcoin and Ethereum support only about 7 and 40 transactions per second (TPS), respectively [40, 12]. As blockchain applications have expanded beyond payments to domains such as decentralized finance and the Internet of Things, ensuring substantially higher throughput has become critical. Early blockchains used monolithic architectures that tightly coupled transaction dissemination, consensus, verification, and execution, making it difficult to optimize individual components. This limitation led to lazy blockchain architectures [14, 25, 56], which separate these functions into distinct stages that can be optimized independently and executed concurrently. As a result, system performance is bounded by the slowest stage. While research has mostly focused on consensus [29], recent approaches, e.g., high-throughput Directed Acyclic Graph (DAG)-based protocols [35, 14, 5, 48], have shifted the main performance bottleneck to the execution layer.

Execution performance can be optimized in two ways. Vertical optimization improves the execution of individual transactions, often by analyzing smart contract languages [26, 2], but its effectiveness depends on transaction type. We focus on horizontal optimization, which increases throughput by executing independent transactions in parallel across multiple executors [39, 3, 52, 8, 18, 15, 27]. Using horizontal optimization, state-of-the-art execution engines reportedly achieve up to 170kTPS [27]. The key property that enables efficient horizontal parallelization is transaction independence. Independent transactions can be executed concurrently across distributed workers, necessitating only a final state merge. In the object model [20, 6, 22], transactions consist of read and write operations on specific state objects and two transactions are therefore considered independent if their operations on common objects are read-only.

Traditional blockchains, such as Bitcoin and Ethereum, allow validators to choose transaction order within a block. This acts as a form of congestion control, as validators prioritize transactions with higher gas fees to mitigate spam and balance throughput with available execution resources [7]. However, this opaque discretion also enables Maximal Extractable Value (MEV), where validators reorder transactions for profit [13]. In response, fair ordering protocols have been proposed for the dissemination and consensus layers [44, 37, 60], enforcing transaction order based on arrival time. These approaches define two extremes: unrestricted validator control and rigid ordering that ignores validator incentives.

In this work, we investigate a middle ground based on the transaction sequencing framework of Sui [24, 23], which uses the object model. In this framework, transactions are either ordered for execution or selectively deferred to be reprocessed with later blocks when resource limits are exceeded. Sui orders transactions by gas price and processes them sequentially, deferring any transaction whose objects would exceed a protocol-defined execution capacity. We extend Sui’s approach by incorporating execution-time estimates [17, 49] and object-conflict information [10, 57] directly into the sequencing process. We specify a sequencing framework that supports the continuous execution model, where the processing of consecutive blocks by different blockchain layers might overlap, and changing congestion conditions. To address this specification, we propose an anytime genetic algorithm¹ that continuously refines the ordering as time permits.

¹ An anytime algorithm returns a valid solution even if interrupted before completion.

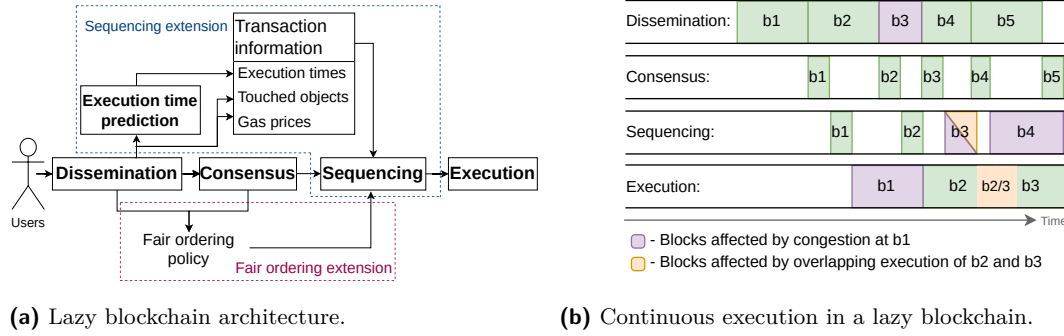


Figure 1 Overview of the sequencing layer in a lazy blockchain. Fig. 1a illustrates the modular architecture we consider, where sequencing bridges consensus and execution by computing schedules based on transaction gas prices, object sets, and predicted execution times, with optional fair ordering constraints. Fig. 1b shows the pipelined processing of blocks across blockchain layers, including two cases where the continuous execution requirement affects sequencing. First, congestion detected during the execution of block 1 causes the dissemination of block 3 to end early, freeing time for additional sequencing of upcoming blocks 3 and 4. Second, an overlap between the execution of blocks 2 and 3 forces early termination of block 3's sequencing to maintain execution continuity.

As a summary, we make the following **contributions**:

- We introduce the first formalization of the transaction sequencing problem within a continuous execution framework. Unlike traditional single-shot approaches, our model explicitly accounts for overlapping block boundaries, transaction object conflicts, and dynamic scheduling deadlines under network congestion. We extend this model to formalize how order-fairness policies can be integrated as optional execution constraints.
- We design and implement an anytime genetic algorithm tailored for the sequencing layer to optimize validator profit using gas prices, object sets, and predicted execution times. The algorithm accommodates real-time execution constraints by immediately outputting a high-quality intermediate schedule if interrupted prematurely. We augment our genetic framework to support optional consensus-determined fair ordering. By strictly enforcing arrival-sequence constraints only for interdependent (conflicting) transactions, our algorithm intelligently allows the concurrent reordering of independent transactions.
- We implement our algorithm and compare it against several sequencing heuristics using real-world Sui data and synthetic Ethereum-based workloads. Our experiments quantify the impact of perturbed execution-time estimates. Using real-world blockchain data, our results demonstrate that the proposed genetic sequencer increases validator profit by up to 16% under sustained congestion and accelerates congestion relief by up to 58% during sporadic transaction spikes.
- We provide the first empirical evidence that enforcing strict fair ordering severely limits independent transaction optimization, imposing a 50% to 60% profit penalty on validators during peak congestion. Based on our model, algorithms, and datasets, our results suggest that rigid fair ordering may eliminate the performance benefits of advanced parallel sequencing layers.

2 Background

2.1 Sequencing

Fig. 1a illustrates the modular architecture of a so-called lazy blockchain, where consensus and execution are decoupled. We focus on the sequencing layer, which receives blocks² from the consensus layer. The sequencing layer assigns transactions to execution workers, manages congestion, and may defer transactions that cannot be completed within the available execution window. We assume that execution-time estimates are available for all transactions. These estimates can be derived from transaction properties as early as during the dissemination phase, when transactions become known, although the estimation method itself is outside the scope of this work. We also consider an optional fair ordering constraint, which requires the sequencing layer to preserve the causal order of transactions. The sequencing layer produces an estimated schedule, while the execution layer performs the actual execution. Because execution times may vary and conflicts can introduce additional delays, the realized execution order may differ from the estimated schedule.

2.2 Scheduling Estimation

While we focus on optimizing transaction ordering, evaluating a sequencing algorithm requires estimating the resulting execution schedule, which ultimately determines execution performance. We assume a scheduling function that maps an ordered set of transactions to estimated start times and worker assignments. Although our method is independent of its specific implementation, we use a concrete scheduler to evaluate results.

Because optimal scheduling is NP-hard, we assume a deterministic polynomial-time greedy scheduler that processes transactions sequentially, assigning each to the earliest feasible position on a worker and deferring those that cannot be safely placed within a worker’s schedule. Our approach extends Sui’s to a fixed number of workers, whereas the original design assumed unbounded parallelism and focused solely on object availability. Alg. 1 details our extended scheduling algorithm.

We use the Earliest Start Time policy, which assigns each transaction to the worker that allows the earliest execution and is the default policy in current blockchains. We further improve scheduling by allowing transactions to be inserted into gaps in existing schedules rather than only appended. While this improves utilization [30], it increases complexity since gap structures must be maintained and queried. For efficiency reasons, we store gaps in a B-tree ordered by duration and search from the smallest feasible gap upward, stopping at the first valid placement. This differs slightly from the original gap-filling formulation but achieves comparable schedule quality with better scalability for large workloads. We therefore adopt this greedy gap-filling scheduler as our evaluation scheduling function.

2.3 Sequencing Objective and Requirements

Objective. We model sequencing as a combinatorial optimization problem that constructs a transaction schedule optimizing a given performance objective. For instance, minimizing makespan targets shortest completion time, while maximizing throughput focuses on the number of transactions executed within a fixed horizon. In this work, we aim at maximizing

² We use the term “block” to denote a set of transactions sequenced and executed together. In DAG-based architectures, a single commit may include multiple vertices.

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    $WrkFreeAt \leftarrow \emptyset$ 
4   for  $w$  in  $0, 1, \dots, m-1$  do           // find earliest free slot per worker
5      $WrkFreeAt[w] \leftarrow \text{find\_earliest}(A, w, tx)$ 
6    $wrk \leftarrow \text{choose\_worker}(WrkFreeAt)$        // choose worker based on policy
7   if  $wrk = \text{None}$  then
8      $A[tx] \leftarrow \text{None}$                        // no free workers
9   else
10     $A[tx] \leftarrow (WrkFreeAt[wrk], wrk)$        // schedule at earliest available slot
11 return  $A$ 

```

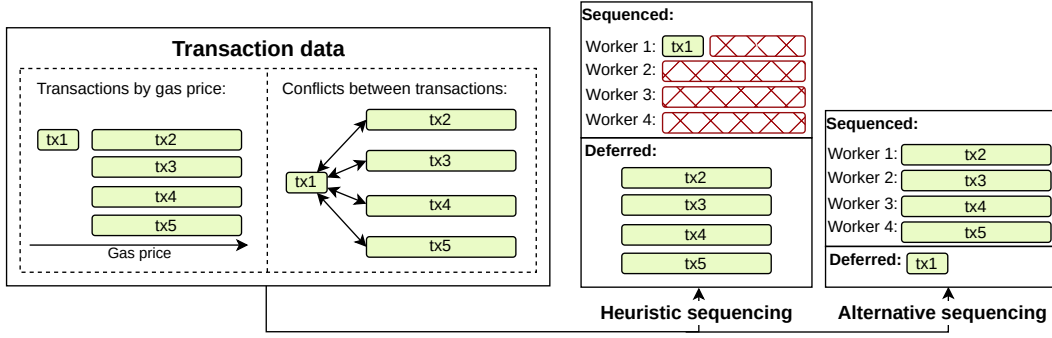
the total fees of scheduled transactions. A transaction's execution fee is calculated by multiplying its execution time by its gas price. This objective prioritizes less resource-intensive transactions while still admitting more congestive ones when their gas price compensates for their cost. It also aligns sequencing with validator incentives by maximizing expected revenue, encouraging continued participation in the network.

Requirements. The sequencing step should satisfy three requirements: high performance, adaptability, and continuous operation. First, since sequencing is meant to improve execution performance, it should not delay it. Second, during low congestion periods, a simple sequencing strategy suffices and spending excessive time on optimization would only increase latency. Differently, in periods of high congestion, a sequencer should dynamically increase its optimization effort and utilize an anytime algorithm to prioritize transaction parallelism, reordering independent transactions while deferring highly conflicting ones. Finally, sequencing must support continuity so execution can proceed to the next block without waiting for the previous one to fully complete. Instead, workers should continuously consume available transactions. As they finish, they immediately take on new work, potentially from later blocks, while earlier transactions may still be executing. This leads to fully overlapping execution across multiple blocks, with no strict barrier between them.

As illustrated in Fig. 1b, these requirements enable cross-layer parallelism. The system can dynamically adjust sequencing effort and timing under congestion, while still maintaining continuous execution and overlapping the processing of successive blocks. Sequencing can be stopped early to immediately feed execution, reducing idle time while allowing later optimization to refine future schedules. We do not assume any specific coordination mechanism between layers. Instead, these requirements are reflected in the design of our ordering algorithm, which must be anytime, and inherently support continuous operation.

2.4 Greedy Ordering Baseline

Sui's current sequencing mechanism uses a heuristic that considers transactions in descending order of their gas prices. Transactions are considered one at a time and either included or deferred based on their estimated impact on shared-object capacity. This greedy approach can perform poorly under contention. A highly congestive transaction may be selected first,



■ **Figure 2** Greedy transaction sequencing based on gas price (left) compared to a feasible alternative (right). The width of each transaction corresponds to its execution time. The greedy algorithm is sub-optimal and defers transactions $tx2-5$, while the alternative algorithm only defers $tx1$.

blocking many smaller transactions that could otherwise execute in parallel. As illustrated in the example of Fig. 2, scheduling transaction 1 first causes transactions 2 to 5 to be deferred, leaving execution workers underutilized. In contrast, deferring transaction 1 and scheduling transactions 2 to 5 increases parallelism, reduces deferrals, and may yield higher gas revenue. Replacing gas price with another single ordering criterion does not resolve this issue, as a highly congestive transaction may still be scheduled first. Instead, we incorporate both execution-time estimates and transaction conflicts into the sequencing process to construct schedules that better utilize available parallelism and improve overall performance.

3 Modeling Continuous Transaction Sequencing

In this section, we present what is, to our knowledge, the first formalization of sequencing within a continuous execution model. We begin by outlining our foundational assumptions, followed by the formalization of both single-shot and multi-shot continuous sequencing. Through these definitions, we explicitly establish the structural constraints that any valid sequencing algorithm, including ours, must satisfy. We introduce our notation throughout the text as needed, and provide a comprehensive summary in Table 1.

3.1 Assumptions

We consider a system of m execution workers on which transactions are scheduled. We assume two conditions: (i) transaction read and write sets are available; and (ii) transaction execution times are known.

The first assumption is directly satisfied in the pessimistic execution model utilized by Sui and Solana, where each transaction must explicitly declare its read and write sets [6, 22]. Even in environments without this structural requirement, static analysis tools can accurately predict or infer these sets beforehand [57, 10]. The second assumption is significantly more challenging to guarantee. Determining a transaction's exact execution path and runtime dynamically is notoriously difficult. Even with pre-execution, the actual processing time can vary based on its final position within a block. Nevertheless, extensive research into historical data analysis and predictive simulation has demonstrated high accuracy in estimating these runtimes [17, 49]. In our experiments we explicitly introduce prediction inaccuracies to evaluate their impact on system performance.

■ **Table 1** Notations.

Symbol	Description
<i>Single-shot sequencing</i>	
d, m	Execution deadline ($d \in \mathbb{R}^+$), Number of workers ($m \in \mathbb{N}^+$)
$T_{\text{all}}, L_{\text{all}}$	Set (T_{all}) and ordered list (L_{all}) of all transactions $\{tx_1, \dots, tx_n\}$
tx_i	Transaction triplet (O_i, t_i, g_i) with execution time t_i and gas price g_i
O_i	Object accesses tuple (R_i, W_i) where $R_i, W_i \subseteq \mathcal{I}$ are read and write sets
σ	Scheduling function: $(L_{\text{all}}, d, m) \rightarrow \Pi_{L_{\text{all}}}$
$\Pi_{L_{\text{all}}}$	Schedule mapping: $tx_i \rightarrow \text{Either}(\text{None}, (s_i, w_i))$ with start time s_i and worker w_i
T_{sched}, P	Scheduled transactions subset ($T_{\text{sched}} \subseteq T_{\text{all}}$) and validator profit ($P \in \mathbb{R}^+$)
<i>Continuous sequencing (at block i)</i>	
$d_i, d_{i_{\text{new}}}$	Base execution deadline and extended deadline
m_i, θ	Number of workers and maximum deferral rounds ($\theta \in \mathbb{N}^+$)
$T_{i_{\text{inc}}}, T_{i_{\text{all}}}, T_{i_{\text{sched}}}$	Incoming, total considered, and scheduled transactions subsets
δ_i, C_i^w	Maximum slack time and latest finish time of worker w
O_{tx_j}	Touched objects set $R_j \cup W_j$ for transaction tx_j
α_k^i, S_i^k	Latest scheduled time for object k , and transactions subset touching object k
<i>Fair ordering</i>	
$L_{\text{fair}}, G_{\text{fair}}$	Total transaction order list, and transaction dependency graph (T_{all}, E)

3.2 Single-Shot Sequencing

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 $L_{\text{all}} = [tx_1, tx_2, \dots, tx_n]$ of n transactions and schedules all or part of them on the m workers over a time period d , producing a schedule $\Pi_{L_{\text{all}}}$.

A set of all transactions $T_{\text{all}} = \{tx_1, \dots, tx_n\}$ 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 $\sigma : (L_{\text{all}}, d, m) \rightarrow \Pi_{L_{\text{all}}}$ is set in advance and represents a scheduler. It takes L_{all} , d and m as input and produces a schedule $\Pi_{L_{\text{all}}} : tx_i \rightarrow \text{Either}(\text{None}, (s_i, w_i))$. 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 $\Pi_{L_{\text{all}}}$ 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 $w_i \in \{0, 1, \dots, m-1\}$. 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 the validator profit $P := \sum_{(O_i, t_i, g_i) \in T_{sched}} t_i \cdot g_i$ where $T_{sched} := \{tx_i \in T_{all} \mid \Pi_{L_{all}}(tx_i) \neq \text{None}\}$ is the set of scheduled transactions.

3.3 Multi-Shot Continuous Sequencing

Our single-shot sequencing formulation captures the transaction sequencing 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. Our goal in this section is to turn this isolated, single-block problem into one whose parameters are shaped by the outcome of the previous block: the only new input carried over between iterations is the initial busy status of the workers, while every other effect of continuity is captured through changes to the problem's parameters. To allow for transaction execution continuity, we extend our formulation of sequencing in several ways.

We label parameters based on the block number for which we generate the sequence: T_{inc} , d_i , m_i are respectively the set of transactions, execution deadline and the number of workers at block i . T_{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 , a parameter 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 $T_{i,all} = T_{inc} \cup \{tx_j \mid \Pi_{L_{i-1}}(tx_j) = \text{None} \wedge tx_j \notin T_{(i-\theta),inc}\}$.

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 , $\delta_i = \max_{w \in [0,1,\dots,m_i-1]} (d_{i,new} - C_i^w)$, 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 cannot 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.

Algorithm 2 Order-fair greedy scheduler.

```

input :  $L_{\text{all}}, d, m$ 
output: Assignment  $A$ , where  $A[tx] = (t, w_{id})$  or None
1  $Def \leftarrow \emptyset$  // deferred txs that will be added to the next block
2  $A \leftarrow \emptyset$ 
3 for  $tx$  in  $L_{\text{all}}$  do
4   if does_conflict_with_any( $tx, Def$ ) then //  $tx$  conflicts with prev. deferred
5      $A[tx] \leftarrow \text{None}$ 
6      $Def \leftarrow Def \cup \{tx\}$  // defer tx
7   continue
8    $WrkFreeAt \leftarrow \emptyset$ 
9   for  $w$  in  $0, 1, \dots, m-1$  do // find earliest free slot per worker
10     $WrkFreeAt[w] \leftarrow \text{find\_earliest}(A, w, tx)$ 
11   $wrk \leftarrow \text{choose\_worker}(WrkFreeAt)$  // choose worker based on policy
12  if  $wrk = \text{None}$  then // no free worker
13     $A[tx] \leftarrow \text{None}$ 
14     $Def \leftarrow Def \cup \{tx\}$  // defer tx
15  else
16     $A[tx] \leftarrow (WrkFreeAt[wrk], wrk)$  // schedule at earliest available slot
17 return  $A$ 

```

3.4 Fair Ordering

So far, our formalization of sequencing can arbitrarily reorder transactions. A strictly opposite approach would enforce fair ordering [37, 60, 53, 51] and focus on maintaining an order of transactions closely aligned with the order in which they have been submitted to the mempool. We extend our continuous sequencing model to include this constraint in order to measure its impact on the performance.

To do so, we modify our models in two ways. First, T_{all} is replaced by $L_{\text{fair}} = [tx_1, tx_2, \dots, tx_n]$ 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 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.

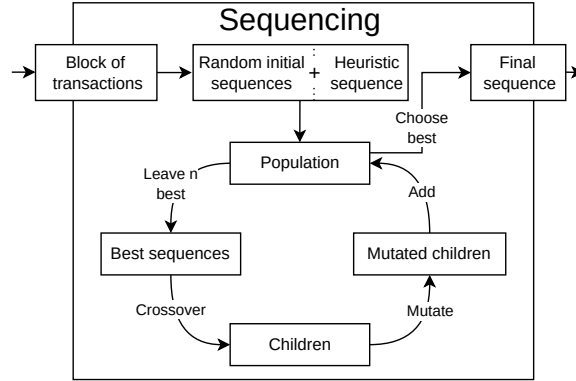
4 Genetic Sequencing Algorithm

Rather than optimizing for an ideal scheduling function, a problem that could be formulated for a constraint solver, we opt against this approach due to its computational inefficiency. As noted by Chahoki et al. [9], using a constraint solver would introduce a new performance bottleneck, ultimately diminishing the throughput benefits of sequencing.

Instead, we focus on transaction ordering under a deterministic scheduling algorithm. This design choice explicitly accounts for imperfect information: the sequencer only has access to execution time estimates, which inherently slightly deviate from actual runtimes in practice, resulting in discrepancies between predicted and actual schedules [17].

Under this deterministic model, the most straightforward approach is to utilize a heuristic to order transactions. Because this problem closely resembles a multidimensional extension of the knapsack problem [19], sorting transactions by gas price serves as our baseline heuristic. For comparison, we also evaluate three alternative heuristics: (i) a randomized order; (ii) the historical arrival order (the sequence in which transactions appeared in the source data); and (iii) shortest execution time first.

4.1 Genetic Ordering



■ **Figure 3** High level overview of our genetic transaction sequencing algorithm.

By fixing the scheduling function, the sequencing problem maps naturally onto a genetic algorithm framework. Here, each candidate solution is encoded as a permutation of the available transactions, with its sequence mapping directly to the scheduler’s processing order. The fitness function is determined by applying the scheduling function and calculating the resulting profit. As illustrated in Fig. 3, the algorithm initializes a population of transaction sequences and iteratively optimizes them through crossover and mutation operations, retaining only the fittest half of the population at each generation.

Because the gas price heuristic often yields high-quality baselines in practice, we seed our initial population with this heuristic ordering rather than starting from a random ordering. Consequently, our approach shifts the genetic algorithm’s objective from an expensive global optimum search to a highly localized refinement of an already strong initial solution.

Alg. 3 details the pseudocode of our genetic sequencing algorithm, which uses seeding, elitism, order crossover (OX) and insertion mutation, which have been shown to be effective for scheduling problems [11]. We discuss these techniques in the following:

■ **Algorithm 3** Basis of our genetic sequencing algorithm.

```

input  :  $budget, pop\_size, T_{all}, \sigma, d, m$ 
output :  $L_{all}$ 

1  $L_{init} \leftarrow \text{highest\_gas\_price\_order}(T_{all})$  // seeding
2  $P_{init} \leftarrow \text{random\_sequences}(T_{all}, pop\_size) \cup \{L_{init}\}$ 
3  $P_{eval} \leftarrow \text{eval}(P_{init}, \sigma, d, m)$ 
4  $budget \leftarrow budget - (pop\_size + 1)$ 
5 while  $budget > 0$  do
6    $P_{kids} \leftarrow \text{generate\_kids}(P_{eval}, pop\_size)$  // OX
7    $P_{kids} \leftarrow \text{mutate}(P_{kids})$  // insertion
8   for  $kid \in P_{kids}$  do
9      $P_{eval} \leftarrow P_{eval} \cup \{\text{eval}(kid, \sigma, d, m)\}$ 
10     $budget \leftarrow budget - 1$ 
11    $P_{eval} \leftarrow \text{order\_by\_profit}(P_{eval})$ 
12    $P_{eval} \leftarrow \text{limit\_population}(P_{eval}, pop\_size)$  // elitism
13 return  $P_{eval}[0]$ 

```

- **Seeding.** An initial solution is added to the starting population. This solution is produced heuristically by ordering transactions based on their gas price [33].
- **OX.** To generate a child from two parent solution sequences, a random slice (random location and random length) is taken from one sequence and is inserted into another sequence at the same spot. Then the resulting sequence is scanned, and the possible second occurrence of a transaction is removed [16].
- **Insertion mutation.** To mutate a sequence, a random transaction is removed and inserted into a different position [31].
- **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 [32].

Seeding and elitism help to ensure faster convergence. To slightly offset this, we found it beneficial to set the mutation probability to 60%. 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. In our experiments, we use $pop_size = 100$, $budget \in \{1000, 5000\}$, $m = 4$. The value we use for parameter d depends on the dataset (62,500 for Sui and 1,250,000 for ETH).

4.2 Adding Fair Ordering

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} . To do so, we apply the following changes. First, we change the initial seed L_{init} to correspond to the sequence obtained through a topological 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, we determine its maximal neighborhood of non-conflicting transactions within the current total order by identifying the closest conflicting transactions before and after it. Then, we relocate the selected transaction to a position that is chosen uniformly at random within this neighborhood.

5 Performance Evaluation

5.1 Settings

In our evaluation, we do not report the runtime of the algorithms, and instead focus on execution performance. The scheduling algorithms are allowed to run until the next block is generated and interrupted if needed. 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. For reference, on the continuous Sui workload, median per-round wall-clock time stayed under 300 ms for all heuristic sequencers, versus roughly 1.0 s for GE10 and 4.4 s for GE50. 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 children 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 1,000 and 5,000 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.

5.2 Datasets and Parameters

The performance of our sequencing approach is highly dependent on the structure and parameters of the transactions to be scheduled. To obtain measurements that are 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.

To evaluate sequencing algorithms, we extract or generate the following parameters for a transaction tx_i :

- Touched objects (O_i) – object set of 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.

The following parameters are independent of transaction data and control how congested the blockchain is. We vary them across experiments to simulate different scenarios.

- Transactions per block ($|T_{inc}|$) – the number of transactions submitted for one block.

■ **Table 2** 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)

- 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.

All transactions are 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.

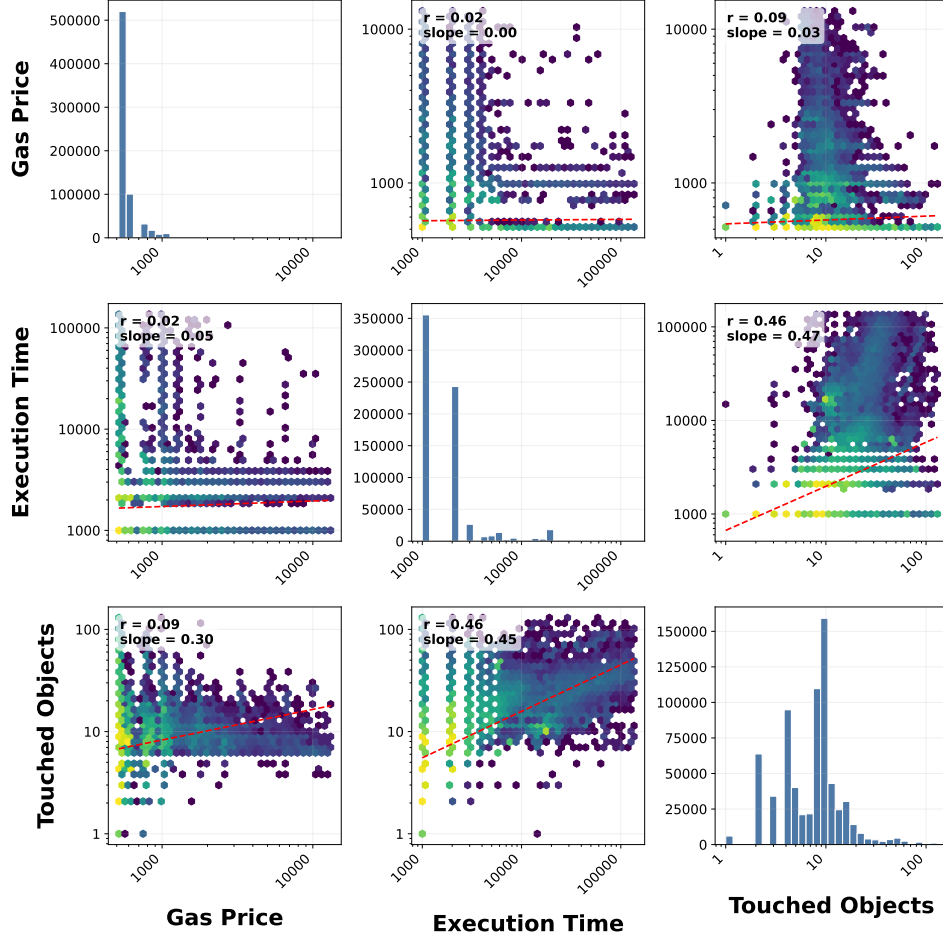
We map platform-specific parameters to the general model as summarized in Table 2. Execution time was derived from transaction effects for Sui and transaction receipt for Ethereum. The gas price was taken from the transaction data for Sui and from the receipt for Ethereum. Touched objects were determined 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 explain below how we add object sets to Ethereum 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 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. Fig. 4 demonstrates the distribution of values for the Sui dataset, and Fig. 5 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 could not infer the accounts that were touched by Ethereum transactions. Instead 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, our Ethereum’s dataset is partially synthetic and we refer to it as the “Ethereum-based” dataset.

As shown in Fig. 6, 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.



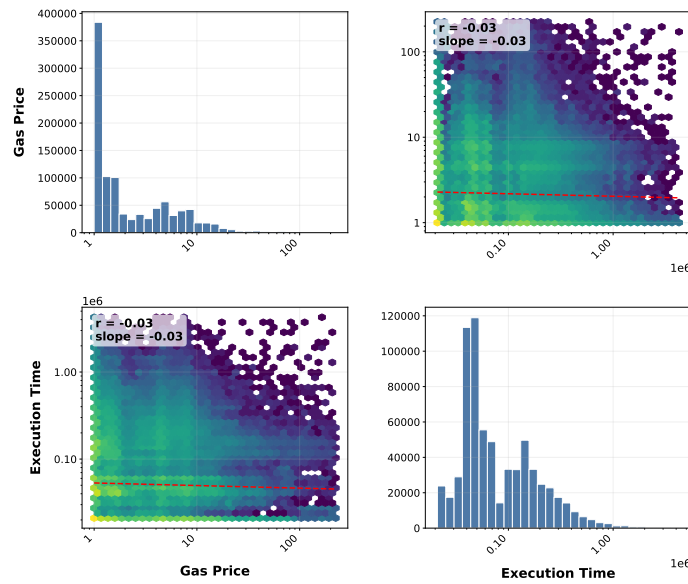
■ **Figure 4** Log-scaled distribution and density plots for gas price, execution time, and total number of touched objects per transaction for SUI data. The units of these metrics are defined by the specific blockchain. Plots include a least-squares regression fit (slope is indicated) and Pearson correlation (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).

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.

5.3 Sequencing Baselines

We evaluated five different sequencing algorithms 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:

- Sui – sequencer that orders transactions by descending gas price and does not use gap-filling.
- Gas Price (GP) – sequencer that orders transactions by descending gas price and does use gap-filling.

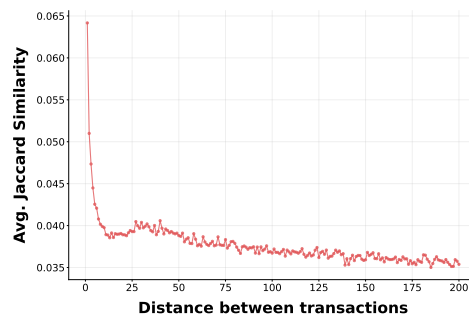


■ **Figure 5** Log-scaled distribution and density plots for gas price and execution time for Ethereum data. Plots include a least-squares regression fit (slope is indicated) and Pearson correlation (r) for each pair of variables.

- Lowest Execution Time (LET) – sequencer that orders transactions by ascending execution time.
- Given – sequencer that orders transactions according to the order in which they have been executed on the blockchain.
- Random – sequencer that shuffles the order of transactions.

We use two versions of our genetic sequencing algorithm with two sets of parameters to demonstrate the impact of search depth on solution efficiency.

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



■ **Figure 6** Average Jaccard similarity between transaction object sets as a function of execution-time distance, computed from a random sample of 50,000 transactions.

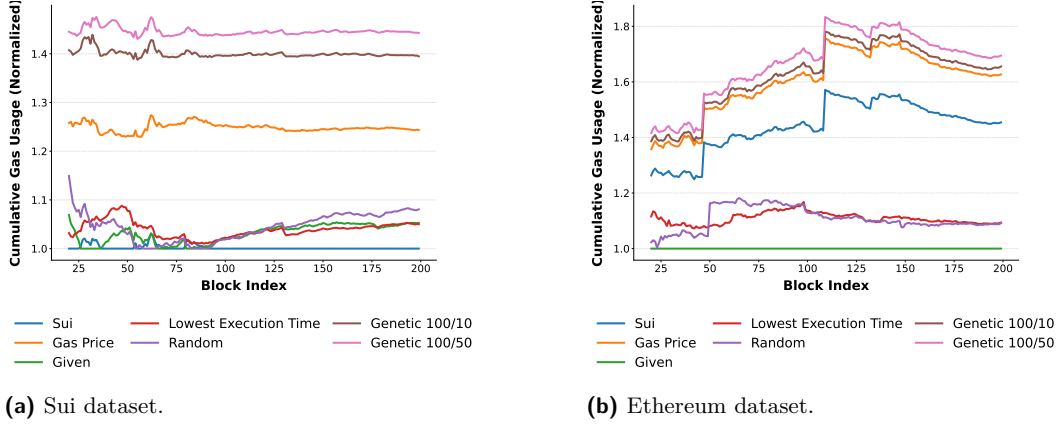


Figure 7 Total normalized validator profit under sustained congestion of 150 transactions per block for 200 blocks starting at block 20.

5.4 Sustained Congestion

Unless stated otherwise, we use $m = 4$ executors, a maximum of 5 deferrals, 150 transactions per block, and an execution deadline of 62,500 time units, yielding an inclusion rate of approximately 50–60% for the GP sequencer. Simulations run for 200 blocks, and we report the cumulative validator profit normalized to the worst-performing sequencer. To highlight steady-state behavior, results are shown from block 20 onward.

Figure 7a presents the results for the Sui dataset. All heuristic baselines except GP achieve similar performance. GP improves validator profit by about 20% over the median heuristic, while the genetic sequencers provide a further 20% gain. Increasing the number of generations from 10 (GE10) to 50 (GE50) yields only a modest additional improvement of roughly 5%, suggesting diminishing returns beyond 10 generations.

For the Ethereum-based dataset, all of the parameters were identical, except for execution time, which was set to 1,250,000 to provide the same 50–60% inclusion rate. The Ethereum dataset results, shown in Fig. 7b, 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 Fig. 4, while Ethereum’s gas prices are much more varied as seen in Fig. 5. 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, because sequencing a high-paying transaction late will have a more severe negative impact on profit. This is also suggested by the Sui sequencer having a much better performance with the Ethereum-based than with 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.

5.5 Transaction Spike

To determine how well our genetic sequencing algorithm works in a scenario where congestion is not sustained but happens in sporadic bursts, we conducted an experiment where initially blocks contain a small number of transactions, then a high number of transactions for some time, and finally a small number of transactions again. We kept the number of executors at 4

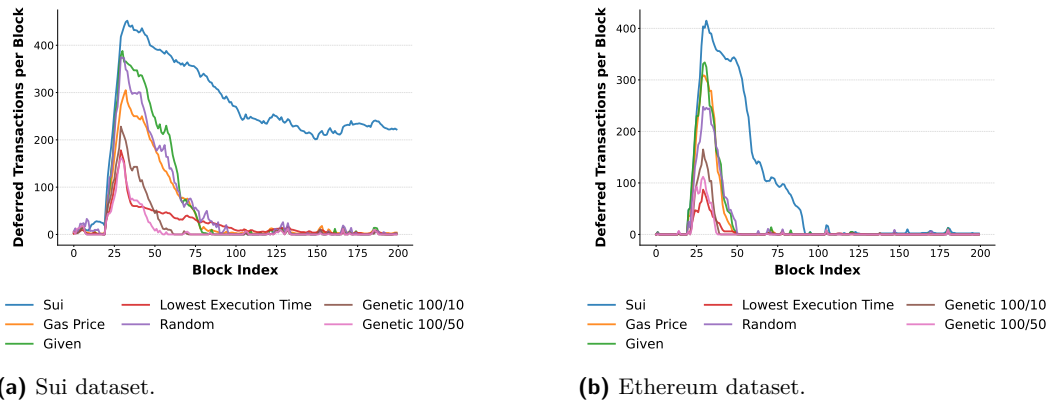


Figure 8 Total absolute number of transactions deferred under a spike in transaction volume from block 20 to block 29. Blocks 0 – 19 contain 30 transactions, blocks 20 – 29 contain 100 transactions, and blocks 30 – 199 contain 30 transactions.

and the execution deadline at 62,500 units for the 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 Fig. 8a. We see greater variations in the results of the heuristic baselines. 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. The LET and Random sequencers recovered around block 90 with the 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 doing it by block 60 and GE10 by block 70.

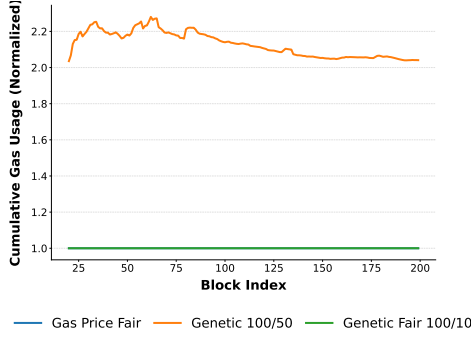
For the Ethereum-based dataset the parameters were the same, with the only difference of execution deadline being, once again, set to 1,250,000 units. The results are shown in Fig. 8b. For the Ethereum-based dataset the congestion was less severe and generally cleared much sooner than in the case of Sui. Besides that, the trend of how quickly sequencers manage to resolve congestion mirrors the Sui results with the exception of the Random sequencer performing on par with the Given and GP sequencers. This is likely because gas prices do not play a role in this scenario.

5.6 Fair Ordering

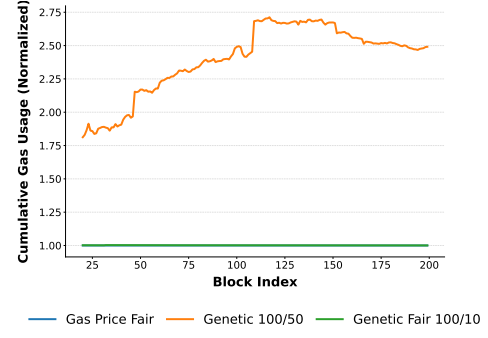
To evaluate the impact of fair ordering, we repeated the sustained-congestion experiment with fair-ordering constraints enabled. Figs. 9a and 9b respectively report the results for the Sui and for the Ethereum-based datasets. For both cases the fair variants of GP and GE10 produce the same results, suggesting 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 sustained congestion, the addition of a fair ordering constraint reduces the profit by 50 to 60%. We attribute this cost to the high rate of transaction interdependence in the dataset (see Fig. 6).

5.7 Robustness to Execution Time Estimation Errors

The strongest assumption in our model is the availability of accurate execution-time estimates. To evaluate robustness to prediction errors, we repeated our experiments with perturbed execution times. Transactions were sequenced using the original execution times as predictions,

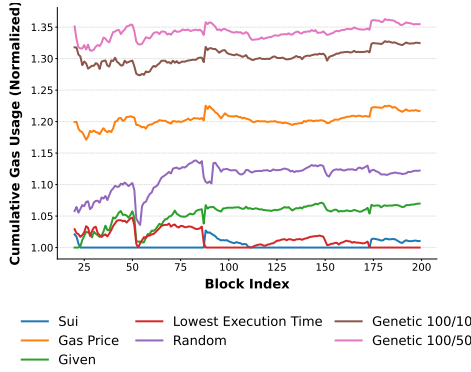


(a) Sui dataset.

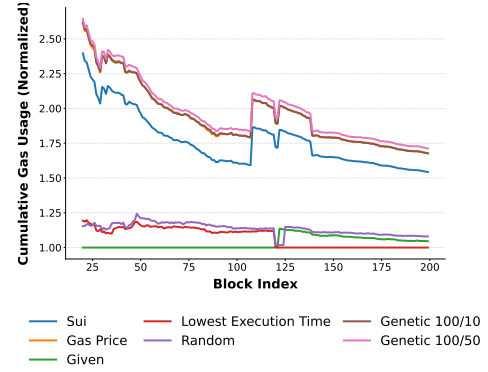


(b) Ethereum dataset.

Figure 9 Total normalized validator profit under sustained congestion of 100 transactions per block for 200 blocks starting at block 20. Sequencers include the fair-ordering schedulers and GE50, the most efficient non-fair scheduler.



(a) Sui dataset.



(b) Ethereum dataset.

Figure 10 Total normalized validator profit under sustained congestion of 150 transactions per block for 200 blocks starting at block 20 with perturbed transaction execution times.

but executed using perturbed values. Perturbations were generated by multiplying each execution time by a log-normal random variable: for each transaction, we sampled $X \sim \mathcal{N}(0, 0.347^2)$ and used e^X as the scaling factor, yielding a 2σ range of approximately $[0.5, 2]$. This configuration matches realistic prediction errors reported by Lima Cabral et al. [17], corresponding to an average absolute error of about 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 Fig. 10a and congestion spike for Sui and Ethereum seen in Figures 11a and 11b showed no statistically significant difference in speed of congestion clearance compared to the non-perturbed case. The sustained congestion for Ethereum seen in Fig. 10b 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.

5.8 Throughput, Latency and Full Summary

In this work, we optimize for validator profit, reflecting the transaction ordering objective used by most existing blockchains, rather than maximizing throughput or minimizing latency. Since we make no assumptions about the underlying hardware or blockchain implementation,

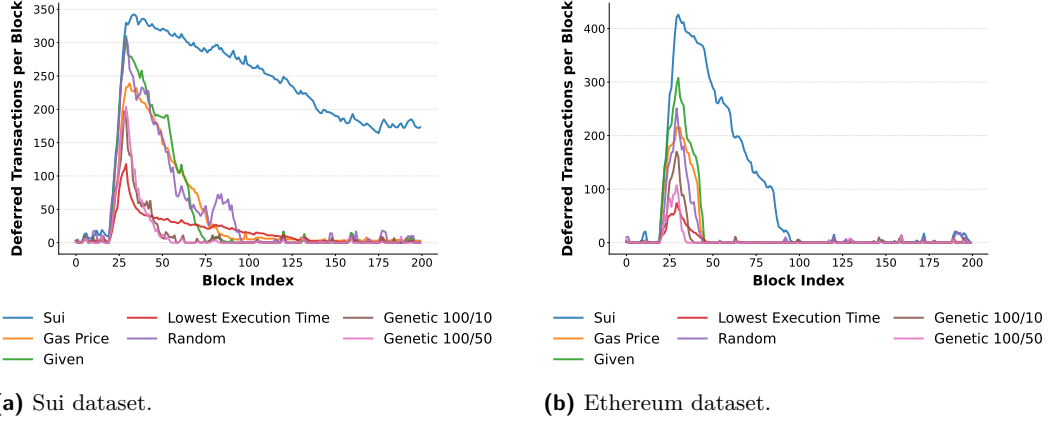


Figure 11 Total absolute number of transactions deferred under a spike (blocks 20 – 29) for Sui and Ethereum datasets with perturbed transaction execution times. Baseline traffic is 30 transactions/block, spiking to 150 transactions/block during the spike.

we report normalized performance metrics. Specifically, throughput is defined as the average number of transactions executed within a time budget d , representing the maximum execution time on a single worker (62,500 units for the Sui dataset and 1,250,000 units for the Ethereum-based dataset).

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

Results are given in Tables 3 and 4 in Appendix A of our extended paper [50]. The genetic sequencer consistently achieves the highest validator profit under sustained congestion, outperforming the GP baseline by respectively 16% and 4% for the Sui and Ethereum-based datasets. For the congestion spike scenario, the genetic sequencer clears congestion 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 consistent results even under perturbation. LET achieves the highest throughput at a significant cost to validator profit while the genetic sequencer consistently improves on the baseline throughput. Latency is broadly comparable across non-fair sequencers, though these figures should be interpreted cautiously, as the assumption of equal sequencing time is unrealistic. Fair ordering imposed a profit penalty of approximately 50% and failed to clear congestion on the Sui dataset.

6 Related Work

We review four related literature areas. First, we examine transaction sequencing in production blockchains, ranging from full validator reordering to protocol-level congestion control. We then consider fair ordering approaches that favor transparency and manipulation resist-

ance over congestion-aware optimization. Next, we review prior work framing sequencing as a discrete combinatorial optimization problem, with an overview provided in Table 5 in Appendix B of our full paper [50]. Finally, we briefly survey work on execution optimization.

Sequencing in production blockchains. In Bitcoin, transaction sequencing within a block was historically of limited importance. The main design goal was to maintain a consistent total order of blocks and ensure immutability unless an adversary controls a majority of hash power. Within a block, ordering was fully determined by the miner. Because Bitcoin uses a UTXO model, each transaction independently defines its inputs and outputs, making intra-block order irrelevant for validity. Only inclusion matters. In practice, transactions are ordered using ancestor fee rate policies, with dependency handling via Child-Pays-for-Parent (CPFP) [43].

Ethereum initially followed a similar approach, but its account-based design makes ordering critical. Transactions read and modify shared state, so execution depends on ordering. This effect is strengthened by smart contracts [7], which allow arbitrary state transitions. Early Ethereum clients such as Geth commonly ordered transactions greedily by gas price [38, 28]. However, Daian et al. [13] showed that validators can increase profit by reordering transactions and inserting their own, a phenomenon known as Miner Extractable Value (MEV). This led to MEV-boost and external block builders, with around 90% adoption by 2022 [47], increasing validator rewards by roughly 250–400% [45, 42] but reducing transparency and decentralization. MEV is commonly extracted through private transaction submission or bundle-based ordering constraints [21].

Sui is a DAG-based system that distinguishes between owned and shared objects, corresponding roughly to UTXO-like and account-like semantics. Owned-object transactions can be partially ordered without global sequencing, while shared objects require global ordering [46]. Originally, conflict resolution for owned objects relied on certification, but in current designs conflicting transactions must be resolved either during consensus or via sequencing. Sui’s sequencing layer also performs congestion control: transactions are first ordered by gas price and then selectively deferred based on estimated execution time and object-level capacity constraints [24, 23].

Solana has also modified its sequencing design in pursuit of high throughput [59]. Early designs used per-thread queues with local priority ordering, which produced partial and non-deterministic global order. Later versions introduced a global priority heap before thread assignment (v1.18). However, sequencing remains largely implementation defined, which has enabled MEV-oriented validator clients such as Jito [41]. This illustrates a recurring pattern: when sequencing is not specified at the protocol level, economic incentives tend to drive systems toward opaque, profit-maximizing ordering strategies.

Fair ordering. The lack of transparency in transaction ordering and the rise of MEV-boosting have motivated research on fair transaction ordering [36]. Its goal is to align transaction order as closely as possible with real submission times. Two main approaches exist. Ordering linearizability, introduced in Pompē [60], uses validator-provided timestamps and orders transactions by the median of a subset of its timestamps. Batch order fairness, as in Themis [37], aggregates full per-validator orderings into a global sequence. Fair ordering improves protocol-level fairness and reduces MEV opportunities, but it imposes a rigid global order, lacks congestion control, and ignores validator incentives.

Combinatorial optimization for sequencing. Under deterministic execution, sequencing reduces to block construction. With sequential execution and position-independent execution times, the problem becomes a knapsack instance and is therefore NP-hard [19]. Karmegam et al. [34] formulate block construction as a mixed-integer program maximizing validator revenue under predefined dependencies and implicit fair-ordering constraints. Their heuristic performs similarly to a gas-price greedy baseline, consistent with our finding that genetic optimization offers little benefit when fairness constraints are enforced. Conthereum [9] models sequencing as a job-shop scheduling problem, optimizing makespan under unrestricted reordering while requiring all transactions to fit within a block. In contrast, we consider continuous execution with overlapping blocks, evaluate robustness to execution-time prediction errors, and study both unconstrained and fair transaction ordering.

Execution. Most prior work targets improvements in the execution layer. Pilotfish [39] is a distributed engine that executes transactions in order while dynamically assigning independent ones to workers, enabling parallelism under a crash-recovery model with low communication overhead. Extensions of Hyperledger Fabric, such as HTFabric and Fabric++ [55, 54], further increase parallelism by reordering transactions during execution. While this improves throughput, it biases against interdependent transactions. Other approaches avoid requiring predeclared object sets. Anjana et al. [4] use software transactional memory (STM) to speculatively execute transactions in parallel with rollback on conflicts. Block-STM [27] refines dependency estimates after rollbacks, reducing redundant execution.

7 Discussion

Gas price. Users do not seem to price the congestion they impose in their transactions, as shown by the weak correlation between gas price and the number of accessed objects (Figs. 4 and 5). Furthermore, we assume a static, user-specified gas price throughout execution. Incorporating dynamic, multi-dimensional pricing mechanisms tailored to parallel execution environments [1, 58] is a promising direction for future work.

Fair ordering constraints and conflict resolution. We derived causal order from total ordering. The observed 50% to 60% profit penalty under sustained congestion highlights the strictness of enforcing this causal chain, at least with the algorithms and datasets we considered. Whether this penalty is verified with all schedulers and datasets is an open question. Alternatively, batch order fairness [37, 51] introduces dependency cycles that could offer scheduling flexibility, though such cycles are rare in practice. Exploring alternative optimization methods like GRASP or simulated annealing remains future work.

External gameability and input resilience. Adversaries cannot game placement via gas bidding without incurring direct execution fees. Furthermore, our evaluation in Section 5.7 demonstrates that our genetic algorithm remains robust against significant execution-time estimation errors. To prevent the case where adversaries declare unused shared objects to trigger inaccurate deferrals, our framework seeds optimization with gas price heuristics to protect higher-paying honest transactions. A possible direction might be the development of native localized fee markets targeting specific shared objects, introducing dynamic fee penalties for declared-but-unaccessed objects or localized pricing mechanisms based on state-declaration accuracy [1, 58].

Real-Time Sequencing. While prior work [9] found genetic algorithms and constraint solvers too slow for online sequencing, our approach uses an anytime algorithm with strict execution deadlines. Seeding the initial population with a gas-price heuristic provides an immediately deployable baseline schedule, while hard execution deadlines interrupt search to ensure continuous worker utilization.

Architectural assumptions. Our model assumes deterministic sequencing over committed consensus batches. While single-leader block builders could act as privileged clients to censor transactions, our target deployment environment relies on DAG-based consensus ([14, 56, 5]). Because transactions are broadcast to multiple nodes and committed to consensus blocks before sequencing, individual validators cannot selectively exclude transactions during optimization without violating state consistency.

8 Conclusion

We have formalized a blockchain-independent model for evaluating the impact of transaction ordering on validator profit and congestion clearance speed. We extended this model to account for continuity of execution and fair ordering. We then proposed a genetic ordering algorithm that takes into account transaction execution time estimates and object conflict information. 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 finally integrated a fair ordering constraint and found that under such an assumption, sequencing provides no benefit. Introducing fair ordering constraints lead validators to lose around 50 to 60% of their profit under sustained congestion.

References

- 1 Bahar Acilan, Andrei Constantinescu, Lioba Heimbach, and Roger Wattenhofer. Transaction fee market design for parallel execution. *arXiv preprint arXiv:2502.11964*, 2025. doi:10.48550/arXiv.2502.11964.
- 2 Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria A Schett. Super-optimization of smart contracts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–29, 2022. doi:10.1145/3506800.
- 3 Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. *CoRR*, abs/1902.01457, 2019. arXiv:1902.01457.
- 4 Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts, 2019. arXiv:1809.01326.
- 5 Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the latency limits with uncertified dags. In *NDSS*, 2025.
- 6 Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. Sui lutris: A blockchain combining broadcast and consensus, 2024. arXiv:2310.18042.

- 7 Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform, 2013. URL: <https://github.com/ethereum/wiki/wiki/White-Paper>.
- 8 Christian Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, volume 310(4), pages 1–4. Chicago, IL, 2016.
- 9 Atefeh Zareh Chahoki, Maurice Herlihy, and Marco Roveri. Conthereum: Concurrent ethereum optimized transaction scheduling for multi-core execution. In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 1–10. IEEE, 2025. doi:10.1109/BRAINS67003.2025.11302903.
- 10 Atefeh Zareh Chahoki and Marco Roveri. Static analysis for detecting transaction conflicts in ethereum smart contracts. *arXiv preprint arXiv:2507.04357*, 2025. doi:10.48550/arXiv.2507.04357.
- 11 Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms—i. representation. *Computers & Industrial Engineering*, 30(4):983–997, 1996. doi:10.1016/0360-8352(96)00047-2.
- 12 Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
- 13 Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE symposium on security and privacy (SP)*, pages 910–927. IEEE, 2020. doi:10.1109/SP40000.2020.00040.
- 14 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22*, pages 34–50, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519594.
- 15 Sourav Das, Vinith Krishnan, and Ling Ren. Efficient cross-shard transaction execution in sharded blockchains, 2021. *arXiv:2007.14521*.
- 16 Lawrence Davis et al. Applying adaptive algorithms to epistatic domains. In *IJCAI*, volume 85, pages 162–164, 1985.
- 17 Danilo Rafael de Lima Cabral, Pedro Antonino, and Augusto Cezar Alves Sampaio. Demystification and near-perfect estimation of minimum gas limit and gas used for ethereum smart contracts. *Journal of Cloud Computing*, 14(1):29, 2025. doi:10.1186/S13677-025-00751-Y.
- 18 Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC '17*, pages 303–312, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3087801.3087835.
- 19 Saulo Dos Santos, Chukwuka Chukwuocha, Shahin Kamali, and Ruppa K. Thulasiram. An efficient miner strategy for selecting cryptocurrency transactions. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 116–123, 2019. doi:10.1109/Blockchain.2019.00024.
- 20 François Ezard, Can Umut Ileri, and Jérémie Decouchant. Nemo: Faster parallel execution for highly contended blockchain workloads. In *2025 7th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 1–5. IEEE, 2025. doi:10.1109/BRAINS67003.2025.11302922.
- 21 Flashbots. Understanding bundles, 2025. Flashbots Documentation. URL: <https://docs.flashbots.net/flashbots-mev-share/searchers/understanding-bundles>.
- 22 Solana Foundation. Sealevel - parallel processing thousands of smart contracts, September 2019.

- 23 Sui Foundation. Object-based local fee markets. Sui Documentation, 2024. Accessed: 2026. URL: <https://docs.sui.io/guides/developer/objects/local-fee-markets>.
- 24 Sui Foundation. Streamlining transactions with sui's shared object congestion control. Sui Blog, September 2024. Accessed: 2026. URL: <https://blog.sui.io/shared-object-congestion-control/>.
- 25 Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1187–1201, 2022. doi:10.1145/3548606.3559379.
- 26 Luciano García-Bañuelos, Alexander Ponomarev, Marlon Dumas, and Ingo 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*, pages 130–146. Springer, 2017. doi:10.1007/978-3-319-65000-5_8.
- 27 Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian 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*, pages 232–244, 2023. doi:10.1145/3572848.3577524.
- 28 go-ethereum contributors. go-ethereum miner/worker.go (commit 290e851), 2017. Accessed: 2026. URL: <https://github.com/ethereum/go-ethereum/blob/290e851f57f5d27a1d5f0f7ad784c836e017c337/miner/worker.go>.
- 29 Yue Hao, Yi Li, Xinghua Dong, Li Fang, and Ping Chen. Performance analysis of consensus algorithm in private blockchain. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 280–285, 2018. doi:10.1109/IVS.2018.8500557.
- 30 Can Umut Ileri, Andrew Cullen, Olivia Saa, Roman Overko, and Luigi Vigneri. Enhanced transaction sequencing for modular distributed ledgers. In *2025 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3. IEEE, 2025. doi:10.1109/ICBC64466.2025.11114535.
- 31 Khalid Jebari, Mohammed Madiafi, et al. Selection methods for genetic algorithms. *International Journal of Emerging Sciences*, 3(4):333–344, 2013.
- 32 Kenneth A. De Jong. *An Analysis Of The Behavior Of A Class Of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.
- 33 Bryant 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, pages 222–226, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/326619.326728.
- 34 Arivarasan Karmegam, Lucianna Kiffer, and Antonio Fernández Anta. Exploiting multi-core parallelism in blockchain validation and construction, 2026. doi:10.48550/arXiv.2602.03444.
- 35 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021. doi:10.1145/3465084.3467905.
- 36 Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In *Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop*, pages 3–14, 2022. doi:10.1145/3494105.3526239.
- 37 Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *Proceedings of the 2023 acm sigsac conference on computer and communications security*, pages 475–489, 2023. doi:10.1145/3576915.3616658.
- 38 Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. Measuring ethereum network peers. In *Proceedings of the Internet Measurement Conference 2018*, pages 91–104, 2018. URL: <https://dl.acm.org/citation.cfm?id=3278542>.

- 39 Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. Pilotfish: Distributed execution for scalable blockchains, 2025. [arXiv:2401.16292](#).
- 40 Wenzheng Li and Mingsheng He. Comparative analysis of bitcoin, ethereum, and libra. In *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, pages 545–550, 2020. doi:10.1109/ICSESS49938.2020.9237710.
- 41 Lostin. The truth about solana local fee markets, January 2025. Helius Blog. URL: <https://www.helius.dev/blog/solana-local-fee-markets>.
- 42 Davide Mancino, Alberto Leporati, Marco Viviani, and Giovanni Denaro. A role and reward analysis in off-chain mechanisms for executing mev strategies in ethereum proof-of-stake. *Distrib. Ledger Technol.*, 4(3), August 2025. doi:10.1145/3672405.
- 43 Johnnatan Messias, Mohamed Alzayat, Balakrishnan Chandrasekaran, Krishna P Gummadi, Patrick Loiseau, and Alan Mislove. Selfish & opaque transaction ordering in the bitcoin blockchain: the case for chain neutrality. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 320–335, 2021. doi:10.1145/3487552.3487823.
- 44 Bulat Nasrulin, Georgy Ishmaev, Jérémie Decouchant, and Johan Pouwelse. Lo: An accountable mempool for mev resistance. In *Proceedings of the 24th international middleware conference*, pages 98–110, 2023. doi:10.1145/3590140.3629108.
- 45 Nero.eth. Is it worth using mev-boost?, June 2024. Ethereum Research Forum. URL: <https://ethresear.ch/t/is-it-worth-using-mev-boost/19753>.
- 46 Roman Overko. A study on shared objects in sui smart contracts. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–7, 2024. doi:10.1109/ICBC59979.2024.10634357.
- 47 Burak Öz, Danning Sui, Thomas Thiery, and Florian Matthes. Who wins ethereum block building auctions and why? *arXiv preprint*, 2024. doi:10.48550/arXiv.2407.13931.
- 48 Nikita Polyanskii, Sebastian Müller, and Ilya Vorobyev. Starfish: A high throughput bft protocol on uncertified dag with linear amortized communication complexity. *IACR Cryptol. ePrint Arch.*, 2025:567, 2025. URL: <https://api.semanticscholar.org/CorpusID:277637515>.
- 49 Miroslav Popovic, Branislav Kordic, and Ilija Basicovic. Estimating transaction execution times for a software transactional memory. In *2016 Sixth International Conference on Information Science and Technology (ICIST)*, pages 137–141. IEEE, 2016.
- 50 Artjom Pugatsov, Can Umut Ileri, and Jérémie Decouchant. The blockchain execution dilemma: Optimizing revenue xor fair ordering. *arXiv preprint arXiv:2604.23266*, 2026. doi:10.48550/arXiv.2604.23266.
- 51 Marko Putnik and Jérémie Decouchant. Herring: Parallel batch-order-fairness on dag-based blockchain consensus. *arXiv preprint arXiv:2605.23648*, 2026. doi:10.48550/arXiv.2605.23648.
- 52 Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 543–557, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389693.
- 53 Giulio Segalini, Yigit Çolakoglu, Marko Putnik, and Jérémie Decouchant. Tilikum: Transaction fair ordering on a dag without weak edges. *arXiv preprint arXiv:2606.27250*, 2026. doi:10.48550/arXiv.2606.27250.
- 54 Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens 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, pages 105–122, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3299869.3319883.
- 55 Jaeyub Song, Juyeong Jeong, Jemin Lee, Inju Na, and Min-Soo 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, pages 2118–2127, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3627673.3679606.

- 56 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIG-SAC Conference on Computer and Communications Security*, pages 2705–2718, 2022. doi: 10.1145/3548606.3559361.
- 57 Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pages 9–16, 2018. doi:10.1145/3194113.3194115.
- 58 Sarisht Wadhwa, Aviv Yaish, Fan Zhang, and Kartik Nayak. Perils of parallelism: Transaction fee mechanisms under execution uncertainty, 2026. URL: <https://eprint.iacr.org/2026/649>.
- 59 Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13, 2018. URL: <https://solana.com/solana-whitepaper.pdf>.
- 60 Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649, 2020. URL: <https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao>.