Exploiting Multi-Core Parallelism in Blockchain Validation and Construction

Arivarasan Karmegam
IMDEA Networks Institute, Madrid, Spain
Universidad Carlos III de Madrid, Spain

Lucianna Kiffer
IMDEA Networks Institute, Madrid, Spain

Antonio Fernández Anta
IMDEA Software Institute, Madrid, Spain
IMDEA Networks Institute, Madrid, Spain

Abstract

Blockchain validators can reduce block processing time by exploiting multi-core CPUs, but deterministic execution must preserve a given total order while respecting transaction conflicts and per-block runtime limits. This paper systematically examines how validators can exploit multi-core parallelism during both block construction and execution without violating blockchain semantics. We formalize two validator-side optimization problems: (i) executing an already ordered block on p cores to minimize makespan while ensuring equivalence to sequential execution; and (ii) selecting and scheduling a subset of mempool transactions under a runtime limit B to maximize validator reward. For both, we develop exact Mixed-Integer Linear Programming (MILP) formulations that capture conflict, order, and capacity constraints, and propose fast deterministic heuristics that scale to realistic workloads.

Using Ethereum mainnet traces and including a Solana-inspired declared-access baseline (Sol) for ordered-block scheduling and a simple reward-greedy baseline (RG) for block construction, we empirically quantify the trade-offs between optimality and runtime. MILPs quickly become intractable as heterogeneity or core count increases, whereas our heuristics run in milliseconds and achieve near-optimal quality. For ordered-block execution, heuristic makespans are typically within a few percent of the MILP solutions (and can even surpass the MILP incumbent when the solver times out), yielding up to 1.5 speedup with $p = 2$ and 2.3 speedup with $p = 8$ over sequential execution, despite tight ordering constraints. For block construction, the heuristic achieves 99–100% of the MILP optimum reward on homogeneous workloads, and 74–100% of an LP-relaxation upper bound on heterogeneous workloads, where exact optimization often times out. The resulting block-construction throughput scales close to linearly with p, reaching up to 7.9 speedup with $p = 8$ in our experiments. These results demonstrate that lightweight, conflict-aware scheduling and selection can unlock substantial parallelism in blockchain validation, bridging the gap between sequential execution and the true potential of multi-core hardware.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Computing methodologies → Parallel algorithms; Computing methodologies → Concurrent algorithms

Keywords and phrases Block construction, Block execution, Deterministic parallelism, Conflict-aware scheduling

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.23

Related Version *Previous Version*: `https://arxiv.org/abs/2602.03444`

Supplementary Material *Software (Source Code)*: `https://github.com/arivarasanka/blockchain_parallel_validation`, archived at `swh:1:dir:0dbd602b6c785657eb927ec026531a7df5c489cc`

Funding This work is funded in part by MICIU/AEI /10.13039/501100011033/, ERDF, and the ESF+ under predoctoral training grant PREP2022-000373 and grant DRONAC (PID2022-140560OB-I00) and by MICIU/AEI/10.13039/501100011033 under grant CEX2024-001471-M.

Acknowledgements We would like to thank Lioba Heimbach and the DISCO group at ETH Zurich for providing the transaction traces used in our experiments section.

1 Introduction

1.1 Motivation

Blockchains process a continuous stream of user transactions that modify a shared replicated state. In most deployed designs (e.g., Ethereum-style execution), validators[1] validate and execute blocks sequentially following a *fixed total order*. While this simplifies reasoning about state, it underutilizes modern multi-core CPUs. In practice, many transactions affect disjoint sets of accounts or storage slots and can be safely executed in parallel without altering the final state. Sequential execution, therefore, limits throughput and increases time-to-finality (i.e., the delay from block proposal to having its transactions finalized on-chain). However, parallelizing block execution is not trivial, since the blockchain state is shared. Reordering or executing concurrently two transactions that conflict[2] when accessing the state may change the final state. Any viable solution must therefore preserve deterministic, order-equivalent semantics while exploiting conflict-free concurrency.

There is also an economic dimension when validators build blocks. They select transactions from a *mempool* of pending transactions with heterogeneous runtimes and rewards. Since blocks are limited in size by execution time (e.g., with a *gas budget* or *runtime limit*), the question for validators is not "how fast can we execute a given block?" but "how do we select the transactions in a block to maximize the reward?"

In this paper, we study how a validator can leverage multi-core hardware without violating blockchain semantics or economic constraints. We study two complementary validator-side optimization problems, one with a fixed block order and one with flexible selection:

Ordered-Block Scheduling (OBS)

Given a block with a fixed transaction set and total order, how should a validator exploit its cores to minimize total execution time while guaranteeing equivalence to sequential execution? This is a **scheduling** problem: transactions that access disjoint sets of accounts or storage slots can run together, while conflicting transactions must respect the block order. We capture conflicts through a **dependency DAG** and aim for a parallel schedule that minimizes the time to execute the whole block (*makespan*), leveraging the inherent parallelism present in existing blocks. This is a classical scheduling problem known as Precedence-Constrained Scheduling, known to be NP-hard on 3 cores and on 2 cores when transactions have execution times in {1,2}[15].

Parallel-Block Construction (PBC)

On the other hand, when assembling a new block under a runtime limit, the validator has to choose from a mempool of pending heterogeneous transactions that differ in both reward and execution time. Here, the problem is deciding which transactions should be included, and how should they be assigned to cores, so that (i) conflicts are respected, (ii) the overall schedule fits within the runtime limit, and (iii) the total validator reward is maximized.

[1] Nodes responsible for proposing or verifying blocks, and executing transactions.
[2] Both write, or one writes and the other reads, the same account or storage slot.

Unlike with OBS, we can now trade off which transactions to include against how efficiently they can be scheduled in parallel. This couples selection and scheduling, reflecting the real economic and system constraints that a validator faces in practice. To our knowledge, prior work has not studied this specific combination of conflict constraints, runtime limit, and validator reward maximization. Observe that PBC is NP-hard, since it generalizes classical NP-hard problems such as knapsack with conflict graphs, and parallel-machine scheduling with conflicts.

1.2 Contributions

(1) **Problem definitions:** We **formalize two validator-side optimization problems:** (i) Ordered-Block Scheduling (OBS), scheduling an ordered block on p cores minimizing makespan, and (ii) Parallel-Block Construction (PBC), selecting and scheduling a subset of mempool transactions on p cores under a runtime limit B to maximize validator reward, all while respecting read/write conflicts.

(2) **MILP formulations:** We present exact **Mixed-Integer Linear Programming (MILP) formulations**[3] that capture conflicts and precedence, core capacity, and budget constraints. These MILPs provide optimal baselines on small-to-medium instances, but become intractable at scale; we therefore use them primarily for benchmarking and insight.

(3) **Fast heuristics:** We design **deterministic heuristics** that produce feasible schedules rapidly, trading optimality for orders-of-magnitude higher speed. E.g., the heuristic for OBS is guaranteed to find a schedule with makespan at most $2 - 1/p$ times the optimal.

(4) **Empirical study:** We quantify empirically the limits of parallelism in block execution and construction. Using Ethereum mainnet traces, we compare MILPs and heuristics on both the time taken to find a solution and its quality. We include in the comparison a Solana-inspired declared-access baseline (Sol) for OBS and a simple reward-greedy baseline (RG) for PBC, instantiated using the access sets extracted from Ethereum traces.

We observe that using multiple cores can significantly reduce makespan and increase the block reward relative to sequential block execution. For instance, for OBS we observe that with only 2 cores we can reduce the makespan by a factor of 1.57, and with 8 cores by more than 2. For PBC, the throughput can be increased almost linearly with the number of cores, and the reward increased by a factor of more than 2 even with 4 cores. We also observe that the runtime for solving the MILPs grows steeply with transaction heterogeneity, core count, and runtime limit. However, our heuristics achieve makespans and rewards close to MILP optima when available (and remain competitive against LP-relaxation upper bounds when exact optimization times out), while running in milliseconds.

1.3 Structure of the Rest of the Paper

The article is structured as follows: Section 2 presents in detail our model and problems statements. Section 3 discusses the heuristics for OBS. Section 4 presents the MILP formalization of PBC and discusses the heuristics for this problem. Section 5 describes the experiment details. Finally, Section 6, presents and discusses the experimental results, while Section 7 concludes the paper. A section with related work is deferred to Appendix A due to space constraints.

[3] Some formulations use only integer or Boolean variables. We refer to all of them as MILP for simplicity.

2 System Model and Problems Definitions

2.1 Blockchains

We model a blockchain as a distributed ledger that maintains a shared state and processes a continuous stream of client-submitted transactions. Pending transactions reside in a *mempool* until they are selected for inclusion in a block. Each block is proposed by a designated *validator* (or leader) that assembles transactions from the mempool, subject to an upper bound B on block execution time.

2.2 Transactions

We use the transaction model of Ethereum. Each transaction τ has execution and fee parameters that govern how it is processed by the validator. When executed, τ consumes a certain amount of gas $\tau.gasU$. It also has a *Priority Fee*, denoted $\tau.tip$, so that $\tau.reward = \tau.gasU \cdot \tau.tip$ is the reward to the validator that includes τ in a block.

We consider two execution-time regimes:

- **Homogeneous** transactions with identical execution time (equivalently identical $\tau.gasU$).
- **Heterogeneous** transactions with varying execution times (equivalently varying $\tau.gasU$).

In Ethereum-like settings, simple transfers are often homogeneous, whereas smart contract transactions are typically heterogeneous. We assume that when a transaction τ is issued, the client includes an *access list* specifying the set of state keys (accounts and storage slots) $\tau.R$ and $\tau.W$ that τ reads and writes, respectively. In practice, transfers typically access only a few keys (e.g., sender and receiver of cryptocurrency), whereas smart contract transactions may access many keys.

Conflicts

We say that two transactions τ_i and τ_j conflict if and only if they access a common state key and at least one of them writes to it. I.e., read–read overlaps are not conflicts, but any overlap involving a write is. Formally,

$$\text{conflict}(\tau_i, \tau_j) \equiv \Big((\tau_i.W \cap \tau_j.W) \cup (\tau_i.W \cap \tau_j.R) \cup (\tau_i.R \cap \tau_j.W) \Big) \neq \varnothing. \tag{1}$$

2.3 Problem Definitions

Our goal is to exploit multi-core hardware to validate and execute transactions efficiently while preserving blockchain semantics. We assume a validator has an execution environment that can process up to p transactions in parallel, where p is the number of CPU cores of the validator. We also assume that the time to execute a transaction τ is proportional to its gas consumption, which we model as exactly $\tau.gasU$. Then, we can define:

▶ **Definition 1** (Schedule). *A schedule S of a set of transactions T on p cores assigns to each transaction in T a core and a starting time to run. Given a schedule S of a set of transactions T on p cores, its* makespan*, denoted $S.mspan$, is the time to complete the execution (without preemption) of all transactions in T as defined by the schedule S.*

We now formalize the two problems.

▶ **Problem 1.** ***Ordered-Block Scheduling (OBS):*** *Given a block with the sequence of transactions $T = (\tau_1, \ldots, \tau_n)$, find a schedule S of the transactions in T on p cores, that minimizes makespan $S.mspan$, subject to the constraint that conflicting transactions are not executed in parallel, and they are executed in the order of the sequence.*

▶ **Problem 2.** ***Parallel-Block Construction (PBC):*** *Given a runtime limit B and a set of transactions in the mempool $Mpool = \{\tau_1, \ldots, \tau_n\}$, select a subset of transactions $T \subseteq Mpool$ and a schedule S of T on p cores such that: conflicting transactions are not executed in parallel, $S.mspan \leq B$, and the total reward $T.reward = \sum_{\tau \in T} \tau.reward$ is maximized.*

3 Ordered-Block Scheduling (OBS)

In this section we propose algorithms to solve the Ordered-Block Scheduling (OBS) problem. All solutions start by creating a dependency graph. Given a block containing an ordered sequence of transactions $T = (\tau_1, \ldots, \tau_n)$, we construct a **directed dependency graph** $G = (T, E)$, where each node represents a transaction and each directed edge $(i, j) \in E$ indicates that transaction τ_i must precede τ_j. An edge exists if (i) the two transactions conflict (as defined in Equation (1)), and (ii) $i < j$, ensuring consistency with the block's total order. This graph captures all read/write dependencies that restrict concurrent execution. Due to space constraints, we defer the MILP formulation for OBS and its discussion to Appendix B.

Our heuristics (Algorithms 4–7, deferred to Appendix C) use a conflict-aware greedy scheduler, guided by transaction priorities. It proceeds in three stages:

(1) DAG construction. For the given block $T = (\tau_1, \ldots, \tau_n)$, we build a dependency DAG $G = (V, E)$ (Algorithm 4). Each node is a transaction $\tau \in T$, and we add an edge (τ_i, τ_j) whenever τ_i and τ_j conflict (as defined in Equation (1)) and $i < j$. This DAG encodes all precedence constraints that must be respected at execution time.

(2) Priority scoring. We then extract structural information from G (Algorithm 5). For each transaction τ, we compute:

- $hgt[\tau]$: the critical-path length rooted at τ (i.e., τ's own execution time plus the maximum downstream height),
- $vol[\tau]$: the total execution volume of τ and its descendants,
- $|succ[\tau]|$: the fan-out of τ.

A single backward pass over the DAG computes these values. We assign each transaction a deterministic priority key $pri[\tau] := \big(-hgt[\tau],\ -vol[\tau],\ -|succ[\tau]|,\ \tau.id\big)$, which favors long critical paths first, then heavier subtrees, then higher fan-out, and $\tau.id$ breaks ties.

(3) Transaction scheduling. Finally, we simulate the block execution on p cores (Algorithm 6). The scheduler maintains (i) a set of *ready* transactions (initially those with in-degree zero in G), (ii) a set of *running* transactions with assigned cores and finish times, and (iii) the current time *now*. Whenever any core becomes free, we schedule the highest-priority ready transaction to that core. When a transaction finishes, we decrement the in-degree of its successors and add any successor with zero in-degree to the set of ready transactions. This event-driven loop continues until all transactions are scheduled. Algorithm 7 combines these steps into a complete pipeline.

$$\max \quad \sum_{i\in[n]} \sum_{r\in[R]} w_i\, x_{ir} \qquad \text{(Maximize total weight)} \tag{2}$$

$$\text{s.t.} \quad \sum_{r\in[R]} x_{ir} \le 1 \qquad \forall i \in [n] \qquad \text{(Each trans. scheduled at most once)} \tag{3}$$

$$\sum_{i\in[n]} x_{ir} \le p \qquad \forall r \in [R] \qquad \text{(At most } p \text{ transactions per round)} \tag{4}$$

$$x_{ir} + x_{jr} \le 1 \qquad \forall (i,j) \in E, i < j,\ \forall r \in [R] \qquad \text{(No conflicting pair in same round)} \tag{5}$$

$$x_{ir} \in \{0,1\} \qquad \forall i \in [n],\ \forall r \in [R] \qquad \text{(Binary assignment)} \tag{6}$$

MILP 1: Parallel Block Construction – Homogeneous Transactions.

Complexity

The DAG construction in Algorithm 4 compares each ordered pair (τ_i, τ_j) with $i < j$, and is therefore $\mathcal{O}(n^2)$ in the worst case.[4] The preprocessing pass in Algorithm 5 performs a single reverse topological traversal plus constant work per edge, i.e., $\mathcal{O}(n + |E|)$. The scheduler in Algorithm 6 runs as an event-driven simulation with at most n schedule events and n completion events; using a priority queue for the ready set, this is $\mathcal{O}\big(n \log n + |E|\big)$. Overall, the heuristic end-to-end runtime is dominated by DAG construction and is $\mathcal{O}(n^2)$ in the worst case. Regarding the approximation ratio, since the heuristic implements a list scheduling, we know that it gives a schedule with makespan at most $2 - 1/p$ times the optimal.

4 Parallel-Block Construction (PBC)

We now turn to the block-construction problem. Given a mempool $\mathit{Mpool} = \{\tau_1, \ldots, \tau_n\}$ of pending transactions, we construct an undirected conflict graph $G = (\mathit{Mpool}, E)$ whose nodes correspond to the transactions in Mpool, with an edge $(\tau_i, \tau_j) \in E$ if and only if the two transactions conflict (as defined in Equation (1)). The block runtime limit B and the conflicts must be respected in the schedule obtained for the transactions selected from T to be included in the block. This setting combines both *selection* (which transactions to include) and *scheduling* (how to assign them across cores) under dependency and time budget constraints.

4.1 MILP Formulations

4.1.1 Homogeneous Transactions

We first consider the homogeneous case, where all transactions have identical execution time t. The block gas budget B therefore allows at most $R = \lfloor B/t \rfloor$ execution rounds. MILP 1 formalizes the selection-and-scheduling problem for this setting. Here, $x_{ir} = 1$ if and only if transaction τ_i is selected and executed in round r, yielding a validator reward $w_i := \tau_i.\mathit{reward}$ for its inclusion in the block.

The problem can be viewed as a weighted packing task: we fill each of the R rounds with up to p non-conflicting transactions, choosing the subset that maximizes total weight (i.e., reward $\sum_i w_i$) while respecting both conflict and capacity constraints.

[4] In practice, this can often be pruned with sparse access lists, but we analyze the dense upper bound.

$$\max \sum_{i\in[n]} w_i\, v_i \qquad \text{(Maximize total weight)} \tag{7}$$

$$\text{s.t.} \sum_{c\in[p]} x_{ic} = v_i \qquad \forall i \in [n] \qquad \text{(One core max)} \tag{8}$$

$$e_i - s_i = t_i v_i \qquad \forall i : i \in [n], \qquad \text{(Time activation)} \tag{9}$$

$$0 \le s_i \le B v_i, 0 \le e_i \le B v_i \qquad \forall i : i \in [n], \qquad \text{(Activation bounds)} \tag{10}$$

$$e_i - s_j \le B\big(1 - y_{ij}\big), \qquad \forall (i,j) : (i,j) \in E \wedge i < j, \qquad \text{(Conflicts)} \tag{11}$$

$$e_j - s_i \le B\big(y_{ij}\big), \qquad \forall (i,j) : (i,j) \in E \wedge i < j, \qquad \text{(Conflicts)} \tag{12}$$

$$w_{ijc} \le x_{ic}, w_{ijc} \le x_{jc} \qquad \forall (i,j) \in \mathcal{P}_{\text{non}}, \forall c \in [p] \tag{13}$$

$$w_{ijc} \ge x_{ic} + x_{jc} - 1 \qquad \forall (i,j) \in \mathcal{P}_{\text{non}}, \forall c \in [p] \tag{14}$$

$$z_{ij} = \sum_{c\in[p]} w_{ijc} \qquad \forall (i,j) \in \mathcal{P}_{\text{non}} \qquad \text{(Same core flag)} \tag{15}$$

$$e_i \le s_j + B\big((1 - y_{ij}) + (1 - z_{ij})\big), \qquad \forall (i,j) \in \mathcal{P}_{\text{non}} \tag{16}$$

$$e_j \le s_i + B\big(y_{ij} + (1 - z_{ij})\big), \qquad \forall (i,j) \in \mathcal{P}_{\text{non}} \tag{17}$$

$$s_i, e_i \in \mathbb{N} \qquad \forall i \in [n] \qquad \text{(Integer assignment)} \tag{18}$$

$$v_i, w_{ijc} x_{ic}, y_{ij}, z_{ij} \in \{0,1\} \qquad \forall i,j \in [n],\ \forall c \in [p] \qquad \text{(Binary assignment)} \tag{19}$$

MILP 2: Parallel Block Construction – Heterogeneous Transactions. Let $\mathcal{P}_{\text{non}} := \{(i,j) : i < j \wedge (i,j) \notin E \wedge (j,i) \notin E\}$ be the set of pairs of transactions without conflicts.

Correctness

Constraint (3) ensures that each transaction is scheduled at most once across all rounds. Constraint (4) enforces the validator's core capacity, limiting each round to at most p concurrent transactions. Constraint (5) excludes conflicting pairs from being assigned to the same round, preserving deterministic execution semantics. Together, these constraints guarantee that every feasible solution corresponds to a conflict-free parallel schedule that fits within the gas budget. The objective (2) then selects the optimal subset of transactions that maximizes total validator reward under these feasibility constraints.

4.1.2 Heterogeneous Transactions

We now extend to the case of transactions with heterogeneous execution times t_i. MILP 2 jointly models transaction selection and parallel scheduling under conflicts, core capacity, and runtime constraints. The binary variable v_i determines whether transaction τ_i is selected to be included in a block. If selected ($v_i = 1$), it must be assigned to exactly one core via x_{ic} and executed during the time window $[s_i, e_i]$ with duration t_i. Conflict edges $(i,j) \in E$ forbid temporal overlap between the conflicting transactions. Their mutual ordering is decided by y_{ij}, enforced through constraints (11)–(12). For non-conflicting pairs, overlap is allowed unless they share a core. Same-core assignment is captured by w_{ijc}, the logical AND of x_{ic} and x_{jc}, and z_{ij} which flags whether a pair shares any core. If $z_{ij} = 1$, the pair must be serialized, with order selected by y_{ij} as enforced in constraints (16)–(17). The objective (7) maximizes the total weight $\sum_{i\in[n]} w_i v_i$ subject to gas and core limits, balancing which transactions are included and how they are scheduled.

Algorithm 1 SchedulePBC().

```
1: function SchedulePBC(T, p, B, pri, confs)
2:     now ← 0
3:     free ← [1..p]                          ▷ Free cores
4:     running ← ∅                            ▷ Trans. running
5:     sel ← ∅                                ▷ Trans. completed
6:     ready ← T                              ▷ Pending transactions
7:     deferd ← ∅                             ▷ Conflicting with running
8:     repeat
9:         while (ready \ deferd ≠ ∅) ∧ (free ≠ ∅) do
10:            τ ← arg min{pri[τ'] :
11:                        τ' ∈ ready \ deferd}
12:            if (running ∩ confs[τ] ≠ ∅) then
13:                deferd ← deferd ∪ {τ}
14:                continue
15:            end if
16:            ready ← ready \ {τ}
17:            if now + τ.t > B then
18:                continue
19:            end if
20:            running ← running ∪ {τ}
21:            sel ← sel ∪ {τ}
22:            start[τ] ← now
23:            end[τ] ← now + τ.t
24:            core[τ] ← any element in free
25:            free ← free \ {core[τ]}
26:        end while
27:        now ← min{end[τ] : τ ∈ running}
28:        S ← {τ ∈ running : end[τ] = now}
29:        running ← running \ S
30:        for each τ ∈ S do
31:            free ← free ∪ {core[τ]}
32:            deferd ← deferd \ confs[τ]
33:        end for
34:    until (running = ∅) ∧ (ready = ∅)
35:    return (sel, start, core)
36: end function
```

Correctness

Constraint (8) ensures that every selected transaction occupies exactly one core, while (9) and the activation bounds couple start and end times to the selection variable v_i. Conflict pairs $(i, j) \in E$ are serialized via constraints (11)–(12), preventing overlapping execution regardless of order. For non-conflicting transactions, same-core co-assignment is detected by w_{ijc} and summarized by z_{ij}; if $z_{ij} = 1$, constraints (16)–(17) enforce a valid ordering determined by y_{ij}. These constraints guarantee that all selected transactions can be executed within the gas limit B without violating conflict or resource dependencies. The objective then identifies the highest-reward feasible subset and schedule.

Complexity

Both the homogeneous (MILP 1) and heterogeneous (MILP 2) formulations of PBC generalize NP-hard problems such as weighted parallel-machine scheduling and multidimensional knapsack. The homogeneous formulation involves $\mathcal{O}(nR)$ binary variables and $\mathcal{O}(nR + |E|R)$ constraints, while the heterogeneous one expands to $\mathcal{O}(n^2p)$ binary variables and $\mathcal{O}(n^2p + |E|)$ constraints. Solver time thus grows sharply with block size and heterogeneity, motivating the use of fast approximation methods.

4.2 Heuristics

Our mempool heuristics (Algorithms 1–3) greedily assemble a conflict-free parallel schedule of transactions under the runtime limit B. They combine lightweight scoring and event-driven dispatch to approximate the MILP objective efficiently.

- **Priority Scores:** For each transaction τ in the mempool, we compute its value density as $\rho[\tau] := \frac{\tau.reward}{\tau.t}$ and $d[\tau]$ is the number of transactions with which τ conflicts. The composite priority key $pri[\tau] := (-\rho[\tau], d[\tau], -\tau.reward, \tau.id)$, favors high-reward, low-conflict transactions, breaking ties deterministically by $\tau.id$.

Algorithm 2 ScoringPBC().

```
1: function ScoringPBC(G)
2:     for each τ ∈ V do
3:         confs[τ] ← {τ' : ((τ, τ') ∈ E)∨
4:                        ((τ', τ) ∈ E)}
5:         d[τ] ← |confs[τ]|
6:         ρ[τ] ← τ.reward/τ.t
7:         pri[τ] ← (−ρ[τ], d[τ], −τ.reward, τ.id)
8:     end for
9:     return (pri, confs)
10: end function
```

Algorithm 3 PBC Heuristics.

```
1: Input: Set T = {τ_1, ..., τ_n}. Number of cores
   p. Time/Gas limit B.
2: Output: A schedule with a list of selected trans-
   actions sel and its start time start[τ] and core
   core[τ] assigned to each sel transaction τ ∈ sel.
3: G ← BuildDAG(T)
4: (pri, confs) ← ScoringPBC(G)
5: (sel, start, core) ←
6:                 SchedulePBC(T, p, B, pri, confs)
```

- **Transaction Scheduling:** Using the priorities above, we iteratively dispatch transactions to free cores (Algorithm 1). At each step, the scheduler maintains three sets: *ready* transactions (eligible for execution), *running* transactions (currently executing), and a *deferred* set of candidates that conflict with running ones. When a core becomes free, the highest-priority non-conflicting transaction is launched, provided the cumulative runtime used remains within B. On each completion event, the corresponding transaction is removed from *running*, its conflicts are cleared from *deferred*, and its core is released. This process continues until either the runtime budget or transaction pool is exhausted. Algorithm 3 integrates both steps into a full end-to-end pipeline.

Complexity

The conflict graph construction has complexity $\mathcal{O}(n^2)$. Computing transaction degrees and priorities has also complexity $\mathcal{O}(n^2)$. The event-driven scheduler then performs $\mathcal{O}(n \log n)$ priority operations plus $\mathcal{O}(np)$ updates for dispatch and completion across p cores. Overall, the heuristic runs in polynomial time, dominated by graph construction and conflict detection.

5 Experiments

As described, we study two validator-side tasks: (i) OBS: scheduling a fixed ordered block on p cores to minimize makespan and (ii) PBC: selecting and scheduling mempool transactions to maximize reward under a runtime budget B. We evaluate our MILP formulations and conflict-aware greedy heuristics on transaction traces extracted from the Ethereum mainnet for blocks 21,631,019 - 21,635,079, spanning 15 - 16 January 2025. Each transaction trace records the execution outcome, including the read and write sets, gas used, and internal call tree produced by the Ethereum Virtual Machine (EVM). From these traces, we extract per-transaction `gasUsed`, `to`/`from` addresses, storage keys accessed, and all contract calls. Conflicts are then derived exactly from overlapping read/write sets (as described in Equation (1)). In addition to comparing our heuristics against the MILP baselines, for OBS, we include a baseline inspired by Solana's declared-access execution model (Sealevel), where transactions expose read/write account sets up front and the runtime schedules only non-conflicting transactions to run concurrently [33, 36]. We denote this baseline by **Sol**. On our Ethereum-derived instances (using the access sets extracted from traces), we implement Sol as a deterministic, event-driven scheduler that scans transactions in block order and dispatches each transaction as soon as a core becomes available and it does not conflict with currently running transactions. For PBC, we instead compare against a simple **reward-greedy** baseline (**RG**) that uses the same non-conflicting dispatch rule but selects candidates purely by reward-based priority (i.e., favoring higher-fee transactions, breaking ties by arrival order), yielding a lightweight selection-and-scheduling baseline under the conflict constraints.

5.1 Hardware and Software

All experiments were run on a Dell PowerEdge R7615 equipped with a single AMD EPYC 9654P (96 cores/ 192 threads) and ≈1.5TiB RAM, running Ubuntu 24.04.3 LTS. We use MATLAB R2025b for the heuristics and invoke HiGHS 1.7.1 through MATLAB's `intlinprog` for MILPs. We do not limit the number of threads: MATLAB and HiGHS may use all available cores. Unless otherwise specified, solver tolerances follow MATLAB defaults. We set a maximum running time of $MaxTime := 6000$ s for the solver. All heuristics are deterministic; repeated runs on the same workload yield identical schedules.

5.2 Workload Construction

Our experiments use real Ethereum transactions to synthesize controlled workloads for both problems. To control workload size across configurations (R or B, p, X), we build synthetic workloads by taking consecutive mainnet transactions in their original order and re-aggregating them, rather than relying on variable on-chain block boundaries.

Trace filtering

We parse transactions and separate them into two categories:

- **Homogeneous** transactions: pure ETH transfers that do not invoke any smart contract, and therefore use the default gas amount of $21,000$ units.
- **Heterogeneous** transactions: the full set of observed transactions, including contract calls with varying performed computations.

Each transaction's gas used $\tau.gasU$ is taken directly from its execution trace, and its execution time $\tau.t$ is normalized proportionally to gas use.

OBS workloads

For each experiment, we collect a sequence of transactions (Homogeneous or Heterogeneous) in their original blockchain order, ignoring actual block boundaries. We then aggregate this sequence into blocks of size determined by either (a) the number of rounds R for homogeneous cases, each round able to process up to p transactions; or (b) the gas budget B for heterogeneous cases, stopping once $\sum_\tau \tau.gasU \approx B$. For each configuration (R or B, p), we construct five consecutive, non-overlapping workloads. This corresponds to simple transactions extracted from 219 blocks ($9,600$ transactions) of mainnet data for the homogeneous case, and 14 blocks ($2,460$ transactions) of mainnet data for the heterogeneous.

PBC workloads

To emulate a validator's mempool, we use a sliding window of transactions. Given configuration (R, p, X) or (B, p, X), where X is the *pool factor*, we select a candidate pool containing approximately X times the transactions (or total gas) that fit in one block:

- Homogeneous case: $R \cdot p \cdot X$ transactions;
- Heterogeneous case: $B \cdot p \cdot X$, where $\sum_\tau \tau.gasU \approx B$.

The corresponding MILP and heuristic then select and schedule a feasible subset for inclusion. For each configuration, we again construct five independent workloads. In practice, this corresponds to simple transactions extracted from $2,989$ blocks ($137,200$ transactions) of mainnet data for the homogeneous case, and 24 blocks ($4,280$ transactions) of mainnet data for the heterogeneous case.

Experiment parameters

For homogeneous transactions, experiment duration is discretized into equal-length rounds, so progress is naturally parameterized by the number of rounds R, with per-round capacity p. For heterogeneous transactions, we parameterize the experiment duration by a continuous runtime limit B, which directly captures feasibility under varying processing times. For homogeneous cases, we evaluate each configuration with rounds $R \in \{10, 30, 100\}$, cores $p \in \{2, 4, 8\}$, and (when applicable) pool factor X. For mempool experiments we use $X \in \{2, 4, 8\}$ in the homogeneous case and $X \in \{2, 4\}$ in the heterogeneous case. For heterogeneous cases, we set the runtime limit to $B \in \{210\text{K}, 630\text{K}, 2100\text{K}\}$. We choose these B values to span small, medium, and large heterogeneous workloads that remain solvable often enough to permit meaningful comparison against MILP baselines; larger budgets rapidly become dominated by solver timeouts. For every R or B,p,X configuration, we run five non-overlapping workloads and report the mean.

5.3 Hypotheses

We evaluate the following hypotheses:

- **H1 (MILP scaling):** MILP runtime grows steeply with problem size and heterogeneity.
- **H2 (Heuristic speed):** The conflict-aware greedy heuristics run orders of magnitude faster than MILPs across all settings.
- **H3 (Heuristic quality):** The heuristics yield parallel schedules that approach the performance of MILP solutions.
- **H4 (Practical speedup):** The heuristics yield parallel schedules with significant speedup with respect to sequential executions.

6 Results and Discussion

We now present results for OBS and PBC under homogeneous and heterogeneous workloads. Unless stated otherwise, each reported value is the mean over five independent instances per configuration. We compare our conflict-aware greedy heuristic (GH) to the MILP baselines and to the additional baselines introduced in Section 5.

6.1 Ordered-Block Scheduling

We first compare the runtime of the MILP solver against GH and Sol for OBS on homogeneous workloads with $p \in \{2, 4, 8\}$ and $R \in \{10, 30, 100\}$. Solver time increases rapidly with both R and p, growing from milliseconds on small instances to thousands of seconds for R=100, p=8, confirming the expected combinatorial scaling (**H1**). GH and Sol, by contrast, remain consistently below 0.1 s even at the largest configuration (**H2**). Table 1 shows that on homogeneous instances, GH matches the MILP solution across all tested (R, p) (**H3**). Sol is close but degrades on larger instances; for example, at R=100, Sol requires 106.2 rounds for p=2 and 118.4 rounds for p=4, compared with 100 rounds for MILP/GH. This indicates that simple in-order dispatch leaves avoidable stalls even when conflicts are known. GH achieves a speedup similar to that of MILP, which is near-linear for these low-conflict homogeneous transfer workloads. This gap reflects the difference between a reactive in-order dispatcher (Sol), which may block on transient conflicts, and our proactive scheduling approach (GH), which pre-computes a conflict-respecting schedule that smooths execution by prioritizing transactions on the critical path.

Table 1 OBS – Homogeneous transactions: Average number of rounds (rnd) and average speedup related to the sequential runtime (in times ×) for MILP solver and greedy heuristic (GH) (identical results) compared to Solana (Sol).

	R = 10		**R = 30**		**R = 100**	
p	**MILP/GH** (rnd/times)	**Sol** (rnd/times)	**MILP/GH** (rnd/times)	**Sol** (rnd/times)	**MILP/GH** (rnd/times)	**Sol** (rnd/times)
2	10/2	10.2/1.96	30/2	30.2/1.99	100/2	106.2/1.88
4	11/3.64	13.2/3.03	30.6/3.92	38/3.16	100/4	118.4/3.38
8	11.6/6.9	15.2/5.26	31.6/7.6	44.6/5.38	100/8	115.2/6.94

Table 2 OBS – Heterogeneous transactions: Heuristics solution makespan M as a percentage (%) of the optimal solution found by the MILP solver. For the values marked with † (columns B=2100K), the MILP hit the 6000 s timeout limit: for p=2, 3 of 5 workloads timed out; for p=8, all 5 did; the reported MILP incumbents are feasible but not guaranteed optimal. The second part of a cell shows the corresponding speedup (in times ×) achieved relative to the sequential runtime.

	B = 210K		**B = 630K**		**B = 2100K**	
p	**GH** (%/times)	**Sol** (%/times)	**GH** (%/times)	**Sol** (%/times)	**GH** (%/times)	**Sol** (%/times)
2	100.49/1.65	103.69/1.59	100.29/1.57	107.56/1.45	98.68†/1.81	117.15†/1.51
4	100/1.82	102.9/1.77	100/1.98	105/1.86	100/2.15	111.03/1.94
8	100/2.26	101.19/2.22	100/2.03	101.13/2	77.67†/2.29	81.77†/2.17

We observe a similar runtime separation on heterogeneous workloads, parameterized by gas budget $B \in \{210K, 630K, 2100K\}$. As B grows, solver runtime escalates by orders of magnitude with both p and B (**H1**), exceeding our 6,000 s execution time limit for the largest configuration. GH and Sol, by contrast, remain sub-second throughout (**H2**). Table 2 confirms that GH's makespans differ from the MILP's best integer solutions by at most a few percent (and in some cases even improve upon the solver's incumbent when the solver exceeds our runtime limit) and also consistently outperforms Sol (**H3**). Overall speedups over 1-core sequential execution are in the ∼1.6–2.3× range (Table 2), confirming **H4** but also highlighting a key limitation of OBS: even with many cores, a fixed consensus order and conflict-induced dependencies cap achievable parallelism.

6.2 Parallel-Block Construction

For PBC on homogeneous workloads, we compare runtimes while varying the pool factor $X \in \{2, 4, 8\}$ and cores $p \in \{2, 4, 8\}$. MILP time again increases steeply with X and p, while GH and RG remain in the millisecond range (**H1,H2**). For homogeneous mempool instances, GH and RG achieve reward values that are consistently close to the MILP optimum across all tested configurations ($X \in \{2, 4, 8\}$, $p \in \{2, 4, 8\}$, and $R \in \{10, 30, 100\}$). This shows that when execution times are uniform, greedy conflict-aware packing is sufficient to recover near-optimal block reward, supporting **H3**. Moreover, RG's performance closely matches GH across the entire configuration grid, indicating that the simple reward-greedy baseline remains competitive on homogeneous workloads. Also, in this homogeneous setting both GH and RG achieve essentially perfect parallel utilization, yielding a speedup equal to the number of cores (i.e., exactly p) across the tested configurations, supporting **H4**.

Table 3 PBC – Heterogeneous transactions: MILP incumbent and GH rewards - $\sum_\tau \tau.reward$ of the block generated as a percentage (%) of the LP-relaxation upper bound on achievable reward reported by the solver. The second part of a cell shows the corresponding speedup (in times ×) achieved relative to the sequential runtime.

B = 210K				
	X = 2		**X = 4**	
p	**MILP** (%/times)	**GH** (%/times)	**MILP** (%/times)	**GH** (%/times)
2	99.81/1.61	99.60/1.84	100/1.9	96.84/1.86
4	99.34/3.27	95.47/3.25	91.89/3.87	88.71/3.88
8	100/4.26	98.83/3.82	66.61/4.33	92.21/7.35

B = 630K				
	X = 2		**X = 4**	
p	**MILP** (%/times)	**GH** (%/times)	**MILP** (%/times)	**GH** (%/times)
2	95.24/1.98	94.84/1.99	54.83/1.94	74.74/1.99
4	77.9/3.39	91.9/3.89	14.67/1.98	75.42/3.96
8	15.27/1.41	81.24/7.44	2.94/1	74.03/7.9

For heterogeneous workloads, where both execution times and rewards vary, exact MILP optimization becomes prohibitive. We therefore normalize both solver and heuristic rewards by the MILP's LP-relaxation upper bound[5]. Table 3 presents these normalized values for pool factors $X \in \{2, 4\}$, runtime limits $B \in \{210\text{K}, 630\text{K}\}$, and cores $p \in \{2, 4, 8\}$. GH schedules achieve between 74–100% of the relaxation bound, while MILP incumbents frequently remain well below 20% of the LP bound at large B or p, having reached the time limit before closing the optimality gap (**H1,H2**). Even in the most challenging settings, GH constructs high-reward, feasible blocks within milliseconds, thereby confirming its scalability and practical robustness (**H3**). We also evaluated the reward-greedy baseline (RG) on the same instances and observed near-identical results to GH in this setting; for clarity, we therefore omit RG from the table. The speedup values obtained are close to the value of p, especially for large B (**H4**), which means that using parallelism in this problem achieves almost linear speedup. Comparing with Table 2, we observe a substantial improvement when transaction selection is possible versus only scheduling a pre-determined block.

We conducted additional conflict-stress experiments with artificially high contention and found that GH is consistently at least as fast as RG and yields higher total reward while scheduling the same number of transactions, with an average improvement of 7.7% over RG. In fact, a simple scenario in which GH will drastically improve over RG is one in which all transactions have execution time and reward 1, and the first B transactions in the mempool conflict with each other, while all the other transactions (at least $p \cdot B$) conflict with the first B but not with themselves. (This scenario may arise if the first B transactions write in the same item x, while the rest reads x.) Then, RG will schedule the first B transaction in sequence, not able to schedule other transactions in parallel. On the other hand, GH will schedule $p \cdot B$ transactions, always executing p transactions in parallel.

[5] The MILP solver used initially reports the LP-relaxation bound, obtained by relaxing integrality.

7 Conclusions

In this paper, we presented a systematic exploration of validator-side parallelism in blockchain transaction execution. We formulated two complementary optimization problems: (i) executing an already ordered block on multiple cores while minimizing makespan, and (ii) selecting and scheduling a subset of mempool transactions under a gas budget to maximize validator reward. For both, we developed exact MILP formulations to serve as optimal baselines, and efficient deterministic heuristics that scale to realistic workloads. We also include a Sealevel-inspired declared-access baseline (Sol) to represent a practical, conflict-avoiding in-order execution strategy.

Our evaluation on Ethereum mainnet traces demonstrates that MILP runtimes grow steeply with problem size, heterogeneity, and core count, confirming the computational hardness of the problems. Our heuristics remain sub-second across all settings, produce solutions that are close to those of the MILP, and reduce makespan and increase reward by a sensible amount with respect to sequential scheduling. Across our experiments, GH consistently matches or improves upon Sol in solution quality, with particularly clear advantages as conflict intensity increases, while maintaining comparable (sub-second) scheduling overhead.

From a systems perspective, our findings show that even simple, conflict-aware scheduling strategies can make effective use of available cores, narrowing the gap between theoretical and practical parallelism in blockchain execution. In particular, the mempool-based formulation provides a path toward constructing blocks that are inherently "parallel-friendly", coupling selection and scheduling to improve both performance and economic efficiency.

Limitations and Future Work

Our evaluation uses the `gasUsed` field from Ethereum traces as a proxy for execution time. This provides a consistent and realistic measure for the ordered-block problem, but it is only an approximation of the actual runtime. In practice, execution time and gas usage may diverge due to hardware, client, or EVM implementation differences. Moreover, in the mempool formulation, reordering transactions could change their effective gas consumption in the case of conflicts. A second limitation is that our current heuristics largely resolve conflicts through local, greedy decisions based on static conflict information. A promising direction is to design more holistic conflict-aware policies that reason globally about contention (e.g., hotspots, conflict clusters, or anticipated serialization) when selecting and scheduling transactions. Also, using confirmed transactions rather than a full archived mempool is a deliberate choice in our evaluation, since post-execution traces let us recover exact read/write sets for conflict construction; by contrast, using archived mempool data sources such as Flashbots' Mempool Dumpster would better capture online arrival dynamics, so studying online mempool replay with only pre-inclusion information is an important direction for future work.

Exploring how to model such non-determinism in execution time while maintaining the safety guarantees provided by conservative gas limits is left for future work. Similarly, an open systems question is how best to maintain and update dependency graphs efficiently as new transactions arrive and blocks are mined. This also suggests online heuristics that update priorities as conflicts evolve, rather than relying on a fixed, one-shot ordering. An especially relevant future experiment is multi-day replay over archived mempool traces (e.g., Flashbots' Mempool Dumpster) to evaluate continuously updated mempool conditions and to measure how often the conflict-stress patterns observed in Section 6 arise in practice. Designing low-overhead data structures or incremental graph-maintenance schemes for this purpose will be essential for bringing these scheduling techniques into production blockchain environments.

References

1 Parwat Singh Anjana, Matin Amini, Rohit Kapoor, Rahul Parmar, Raghavendra Ramesh, Srivatsan Ravi, and Joshua Tobkin. Efficient Parallel Execution of Blockchain Transactions Leveraging Conflict Specifications. In Zeta Avarikioti and Nicolas Christin, editors, *7th Conference on Advances in Financial Technologies (AFT 2025)*, volume 354 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:26, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.AFT.2025.29.

2 Aptos. Aptos codebase. https://github.com/aptos-labs/aptos-core.

3 Aptos. Aptos whitepaper. https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf, 2022.

4 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A true concurrent model of smart contracts executions. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages*, pages 243–260, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-50029-0_16.

5 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A theory of transaction parallelism in blockchains. *Logical Methods in Computer Science*, Volume 17, Issue 4, November 2021. doi:10.46298/lmcs-17(4:10)2021.

6 Dvir David Biton, Roy Friedman, and Yaron Hay. Ethereum conflicts graphed. In *2025 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9, 2025. doi:10.1109/ICBC64466.2025.11114508.

7 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. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 2606–2620, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3658644.3670286.

8 Jacek Blazewicz, Mikhail Y. Kovalyov, and Gunter Finke. Scheduling with incompatible jobs. *Discrete Applied Mathematics*, 30(1):9–28, 1991. doi:10.1016/0166-218X(91)90017-V.

9 Nicolas Brauner, Yves Crama, and Francis Sourd. Scheduling with a conflict graph. *4OR*, 14(4):323–356, 2016. doi:10.1007/s10288-015-0294-9.

10 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, 2025. doi:10.1109/BRAINS67003.2025.11302903.

11 Chandra Chekuri and Sanjeev Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM Journal on Computing*, 35(3):713–728, 2005. doi:10.1137/S0097539700382820.

12 Diem. Diem codebase. https://github.com/diem/diem/tree/main.

13 Daniel W. Engels, David R. Karger, Stavros G. Kolliopoulos, Sudipta Sengupta, R. N. Uma, and Joel Wein. Techniques for scheduling with rejection. *Journal of Algorithms*, 49(1):175–191, 2003. doi:10.1016/S0196-6774(03)00078-6.

14 M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975. doi:10.1137/0204035.

15 Michael R Garey and David S Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

16 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*, PPoPP '23, pages 232–244, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3572848.3577524.

17 Ronald L graham. Bounds for certain multiprocessing anomalies. *Bell system technical journal*, 45(9):1563–1581, 1966.

18 Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

19 Yaron Hay and Roy Friedman. Batch-schedule-execute: On optimizing concurrent deterministic scheduling for blockchains. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 163–174, 2024. doi:10.1109/SRDS64841.2024.00025.

20 Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, and Roger Wattenhofer. Defi and nfts hinder blockchain scalability, 2023. doi:10.48550/arXiv.2302.06708.

21 Klaus Jansen and Levente Porkolab. Scheduling jobs with rejection on parallel machines. *Journal of Scheduling*, 13(3):267–277, 2010. doi:10.1007/s10951-010-0167-2.

22 R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 164–173, 2000. doi:10.1109/RELDI.2000.885404.

23 R. Kotla and M. Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584, 2004. doi:10.1109/DSN.2004.1311928.

24 Daniel Kowalczyk and Roel Leus. An exact algorithm for parallel machine scheduling with conflicts. *Journal of Scheduling*, 20(4):355–372, 2017. doi:10.1007/s10951-016-0482-0.

25 Monad Developer Documentation. Parallel execution. https://docs.monad.xyz/monad-arch/execution/parallel-execution, 2025. Monad executes transactions in parallel within a block while preserving the same final outcome as serial execution, using optimistic execution and re-execution when conflicts are detected.

26 Phablo F. S. Moura, Roel Leus, and Hande Yaman. Compact formulations and valid inequalities for parallel machine scheduling with conflicts. *European Journal of Operational Research*, 325(3):433–443, 2025. doi:10.1016/j.ejor.2025.04.006.

27 Ray Neiheiser and Eleftherios Kokoris-Kogias. Anthemius: Efficient & modular block assembly for concurrent execution, 2025. doi:10.48550/arXiv.2502.10074.

28 Ulrich Pferschy and Joachim Schauer. The knapsack problem with conflict graphs. *Journal of Graph Algorithms and Applications*, 13(2):233–249, 2009. doi:10.7155/JGAA.00186.

29 Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, Cham, 5 edition, 2016.

30 Damien Prot and Odile Bellenguez-Morineau. A survey on how the structure of precedence constraints may change the complexity class of scheduling problems. *Journal of Scheduling*, 21(1):3–16, 2018. doi:10.1007/s10951-017-0519-z.

31 Ankit Ravish, Yaron Hay, Piduguralla Manaswini, Rishabh Jain, Roy Friedman, and Sathya Peri. Efficient scheduling of smart contract transactions via conflict graph coloring. In *2025 IEEE 30th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 91–101, 2025. doi:10.1109/PRDC67299.2025.00019.

32 Kun Ren, Alexander Thomson, and Daniel J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832, 2014. doi:10.14778/2732951.2732955.

33 Solana. Sealevel — parallel processing thousands of smart contracts. https://solana.com/news/sealevel---parallel-processing-thousands-of-smart-contracts, 2026. Accessed: 2026-01-19.

34 Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1–2):70–80, 2010. doi:10.14778/1920841.1920855.

35 J. D. Ullman. Polynomial complete scheduling problems. In *Proceedings of the Fourth ACM Symposium on Operating System Principles*, SOSP '73, pages 96–101, New York, NY, USA, 1973. Association for Computing Machinery. doi:10.1145/800009.808055.

36 Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain. White paper, 2017. Version 0.8.13. URL: https://solana.com/solana-whitepaper.pdf.

A Related Work

Parallel execution of transactional workloads under determinism and conflict constraints naturally gives rise to classical combinatorial optimization problems in multiprocessor scheduling and packing. In this work, blockchain execution serves as a motivating application domain in which such constraints arise explicitly and at scale. Similar constraints for replica determinism have long been studied in dependable and replicated systems, where concurrent workloads must produce identical outcomes across replicas to ensure state consistency [22].

At its core, OBS is a precedence-constrained makespan scheduling problem on identical processors ($P|prec|C_{max}$). Multiprocessor scheduling with precedence and resource constraints is NP-hard in general [35, 14], which explains why exact optimization becomes intractable for realistic block sizes and motivates heuristic or approximate approaches. A standard scalable approach is list scheduling, with $(2 - 1/p)$ approximation ratio [17, 18]. Although this bound applies without additional resource conflicts, it motivates greedy event-driven schedulers as practical baselines when optimal solutions are infeasible. More broadly, the $P|prec|C_{max}$ literature shows that the structure of the precedence partial order can significantly affect complexity and approximability [30, 29]. Our precedence DAGs arise from intersecting a total order with a symmetric conflict relation and do not align with known polynomial-time special cases, inheriting the general NP-hardness. Our goal is thus to understand how simple, fast heuristics behave on realistic conflict structures and workload heterogeneity.

A defining constraint in blockchain execution is fixed-order determinism: correctness requires equivalence to a specific total order, not merely serializability to some order. Similar constraints arise in parallel state machine replication and deterministic transactional systems, where concurrency must respect an externally imposed total order [23, 32, 34]. This motivates modeling concurrency as a precedence-constrained scheduling problem rather than allowing arbitrary reordering.

Several blockchain systems exploit parallelism to accelerate execution of an already ordered block. Speculative frameworks such as Block-STM [16], used in Diem [12] and Aptos [2, 3], execute transactions optimistically in parallel and resolve conflicts via deterministic abort-and-retry against a preset transaction sequence. Related approaches such as BTM [1] use read/write-set hints to reduce aborts when access sets are known in advance. Conthereum [10] is closely related to our OBS setting in that it studies conflict-aware intra-block parallel execution for Ethereum, but it differs in using static-analysis-derived conflict information and does not address our PBC problem of reward-maximizing transaction selection under a runtime budget. Other designs rely on explicit conflict declarations: Solana requires transactions to declare their access list, enabling the runtime (Sealevel [33]) to schedule non-overlapping transactions in parallel while serializing conflicts [36]. Object-centric execution follows a similar philosophy, representing state as objects and tracking object-level conflicts; Sui Lutris, in particular, exploits object-level access metadata to enable parallelism while resorting to consensus only when shared-object conflicts require total ordering [7]. Across these designs (including recent systems such as Monad [25]) parallelism is exploited opportunistically, but remains constrained by the prefix of the total order: a conflicting early transaction can stall later independent ones rather than being bypassed. OBS makes this restriction explicit by allowing any order-equivalent execution schedule, thereby formalizing the gap between opportunistic in-order parallelism and optimal makespan-minimizing execution under limited cores.

$$\min \quad \sum_{r\in[R]} y_r \qquad \text{(Minimize the number of rounds used)} \tag{20}$$

$$\text{s.t.} \quad \sum_{r\in[R]} x_{ir} = 1 \qquad \forall i \in [n] \qquad \text{(Schedule each transaction once)} \tag{21}$$

$$\sum_{i\in[n]} x_{ir} \le p \cdot y_r \qquad \forall r \in [R] \qquad \text{(At most } p \text{ transactions per } r \text{ used)} \tag{22}$$

$$\sum_{t=1}^{r} x_{jt} - \sum_{t=1}^{r-1} x_{it} \le 0 \qquad \forall (i,j) \in E, \forall r \in [R] \qquad (j \text{ cannot be scheduled before } i) \tag{23}$$

$$y_r \ge y_{r+1} \qquad \forall r \in [1..R-1] \qquad \text{(Consecutive round usage)} \tag{24}$$

$$x_{ir}, y_r \in \{0,1\} \qquad \forall i \in [n],\ \forall r \in [R] \qquad \text{(Binary assignment)} \tag{25}$$

MILP 3: Ordered-Block Scheduling – Homogeneous Transactions.

A substantial body of work studies scheduling under conflict graphs. For unit-time jobs, conflict-graph scheduling reduces to graph coloring, while heterogeneous execution times break this equivalence [19]. Empirical studies highlight sensitivity to contention and workload structure [31], and measurements on Ethereum show conflict graphs dominated by hotspots, limiting achievable speedups [1, 20, 6]. Complementary semantic analyses characterize when reordering is safe under read/write effects, abstracting away from optimization objectives such as makespan or resource-aware selection [4, 5]. Closely related classical models consider identical-machine scheduling with incompatibility graphs $P|G|C_{max}$, which is NP-hard in general [8, 24, 9], with exact and MILP-based methods applicable only at moderate scales [26].

PBC builds on these models by jointly selecting and scheduling transactions under a hard runtime budget. It generalizes multiple knapsack [11], knapsack with conflict graphs [28], and parallel-machine scheduling with rejection [13, 21]. Unlike existing block construction pipelines, which are typically driven by greedy fee-based selection (e.g., gas price or fee per gas) and largely ignore parallelizability and interaction effects among transactions [27], our formulation explicitly captures the interaction between transaction selection, conflicts, heterogeneous execution times, and core-level makespan. To our knowledge, deployed block assembly pipelines do not incorporate parallelizability directly into transaction selection in a systematic way, making block construction the primary systems gap our work targets.

B Ordered Block Scheduling (OBS) - MILPs

B.1 Homogeneous Transactions

We first consider the homogeneous case in which all the transactions have identical execution time. Since each transaction takes the same amount of time $\tau.t$, we discretize time into rounds R of uniform length. MILP 3 defines the corresponding MILP formulation. Here, R denotes the maximum number of rounds, which must be an upper bound of the rounds required and has to be provided.

B.1.1 Correctness

In the solution of MILP 3, $x_{ir} = 1$ iff τ_i is executed in round r; and y_r indicates whether the round r is used by any transaction. The objective is to minimize the number of rounds used, minimizing total execution time (makespan).

$$
\begin{aligned}
&\min \quad M && && \text{(Minimize makespan)} && (26)\\
&\text{s.t.} \quad \sum_{c\in[p]} x_{ic} = 1 && \forall i \in [n] && \text{(Each trans. in 1 core)} && (27)\\
&\qquad s_i \geq 0 && \forall i \in [n] && \text{(Non-negative start)} && (28)\\
&\qquad s_j \geq e_i && \forall (i,j) \in E && \text{(Precedence constr.)} && (29)\\
&\qquad e_i = s_i + t_i && \forall i \in [n] && \text{(End time Definition)} && (30)\\
&\qquad M \geq e_i && \forall i \in [n] && \text{(Makespan is max end)} && (31)\\
&\qquad w_{ijc} \leq x_{ic} && \forall (i,j) \in \mathcal{P}_{\text{non}}, \forall c \in [p] && && (32)\\
&\qquad w_{ijc} \leq x_{jc} && \forall (i,j) \in \mathcal{P}_{\text{non}}, \forall c \in [p] && && (33)\\
&\qquad w_{ijc} \geq x_{ic} + x_{jc} - 1 && \forall (i,j) \in \mathcal{P}_{\text{non}}, \forall c \in [p] && && (34)\\
&\qquad z_{ij} = \sum_{c\in[p]} w_{ijc} && \forall (i,j) \in \mathcal{P}_{\text{non}} && \text{(Same core flag)} && (35)\\
&\qquad e_i \leq s_j + B\big((1-y_{ij}) + (1-z_{ij})\big), && \forall (i,j) \in \mathcal{P}_{\text{non}} && && (36)\\
&\qquad e_j \leq s_i + B\big(y_{ij} + (1-z_{ij})\big), && \forall (i,j) \in \mathcal{P}_{\text{non}} && && (37)\\
&\qquad M,\, s_i,\, e_i \in \mathbb{N} && \forall i \in [n] && \text{(Integer variables)} && (38)\\
&\qquad x_{ic} \in \{0,1\} && \forall i \in [n],\ \forall c \in [p] && \text{(Binary assignment)} && (39)\\
&\qquad w_{ijc} \in \{0,1\} && \forall (i,j) \in \mathcal{P}_{\text{non}}, \forall c \in [p] && && (40)\\
&\qquad y_{ij}, z_{ij} \in \{0,1\} && \forall (i,j) \in \mathcal{P}_{\text{non}} && && (41)
\end{aligned}
$$

MILP 4: Ordered-Block Scheduling – Heterogeneous Transactions. Let $\mathcal{P}_{\text{non}} := \{(i,j) : i < j \wedge (i,j) \notin E \wedge (j,i) \notin E\}$ be the set of pairs of transactions without conflicts.

Constraint (22) enforces the per-round parallelism limit. Since the binary variable x_{ir} indicates that transaction τ_i executes in round r, the sum $\sum_i x_{ir}$ counts the number of transactions assigned to that round. At most p cores are available, and hence we require $\sum_i x_{ir} \leq p \cdot y_r$, where $y_r = 1$ only if round r is actually used. Therefore, no round can host more than p concurrent transactions, and unused rounds ($y_r = 0$) cannot schedule any work. Observe that in this case, the solution does not need to assign transactions to specific cores.

Constraint (23) enforces precedence. For every directed edge $(i,j) \in E$, transaction τ_i must complete before τ_j begins. The term $\sum_{t=1}^{r} x_{jt}$ equals 1 if τ_j is scheduled in a round $\leq r$, while $\sum_{t=1}^{r-1} x_{it}$ equals 1 if τ_i is scheduled strictly before round r. The inequality

$$\sum_{t=1}^{r} x_{jt} - \sum_{t=1}^{r-1} x_{it} \leq 0, \quad \forall r,$$

therefore excludes any assignment in which τ_j is placed before τ_i. Together, constraints (22) and (23) ensure that (i) no more than p transactions execute in parallel in any round and (ii) all precedence edges are respected in the resulting schedule.

B.2 Heterogeneous Transactions

We now extend the model to the heterogeneous case, in which a block contains both simple and complex transactions with different execution times. MILP 4 encodes this mixed setting. The total execution time (makespan) found is denoted by M, and B is a parameter that must be set to be larger than any possible M. The objective is to minimize the makespan M subject to the assignment, timing, and ordering constraints.

Each transaction τ_i is assigned to exactly one core via a binary variable x_{ic} and scheduled with start and end times s_i and e_i, respectively. Precedence edges $(i,j) \in E$ enforce dependency order via Equation (29), ensuring that conflicting transactions follow the sequence order. For non-conflicting pairs, overlap is permitted unless the two transactions are assigned to the same core. Same-core assignment is captured by the auxiliary variable w_{ijc} (the logical AND of x_{ic} and x_{jc}), and z_{ij} indicates whether the pair shares any core. If $z_{ij} = 1$, the binary selector y_{ij} determines their order: $y_{ij} = 1$ means τ_i precedes τ_j; otherwise, $y_{ij} = 0$ means the reverse. In Equation (36) and Equation (37), observe the following cases:

- $z_{ij} = 0$: the constraints have no effect due to the runtime limit term B.
- $z_{ij} = 1, y_{ij} = 1$: Equation (37) has no effect, only Equation (36) applies, yielding $e_i \leq s_j$.
- $z_{ij} = 1, y_{ij} = 0$: Equation (36) has no effect, only Equation (37) applies, yielding $e_j \leq s_i$.

These conditional constraints ensure that same-core transactions are serialized.

B.2.1 Correctness

Constraint (27) ensures that every transaction is assigned to exactly one core. Precedence edges $(i,j) \in E$ are enforced by (29), which requires that any dependent transaction τ_j starts only after τ_i has completed. Together, these constraints preserve the total order implied by the dependency graph. For non-conflicting pairs, the coupling of w_{ijc}, z_{ij}, and y_{ij} ensures that same-core pairs are serialized by (36)–(37), while pairs on different cores remain unconstrained. Together, these properties ensure that MILP 4 produces a feasible schedule that respects all dependencies and minimizes the overall makespan M.

B.3 Complexity

Both MILP formulations grow quickly with the number of transactions n and cores p. The homogeneous case (MILP 3) involves $\mathcal{O}(nR)$ binary variables and $\mathcal{O}(nR + |E|R)$ constraints, while the heterogeneous case (MILP 4) expands to $\mathcal{O}(n^2p)$ binary variables, $\mathcal{O}(n)$ integer variables, and $\mathcal{O}(n^2p + |E|)$ constraints. Each formulation generalizes well-known NP-hard multiprocessor scheduling problems, implying that exact solutions become computationally prohibitive as n and p increase.

C Ordered Block Scheduling (OBS) - Heuristic Algorithms

Algorithm 4 BUILDDAG(). Recall the conflict definition from Equation (1).

```
1: function BuildDAG(T)
2:     V ← {τ : τ ∈ T}; E ← ∅
3:     for each i ∈ [1..n − 1] do
4:         for each j ∈ [i + 1..n] do
5:             if conflict(τ_i, τ_j) then E ← E ∪ {(τ_i, τ_j)}
6:             end if
7:         end for
8:     end for
9:     return G = (V, E)
10: end function
```

Algorithm 5 ScoringOBS().

```
1: function ScoringOBS(G)
2:     for each τ ∈ V do
3:         succ[τ] ← {τ' : (τ, τ') ∈ E}
4:         pred[τ] ← {τ' : (τ', τ) ∈ E}
5:         outdg[τ] ← |succ[τ]|
6:         indg[τ] ← |pred[τ]|
7:     end for
8:     Q ← {τ : outdg[τ] = 0}                ▷ Set of sinks
9:     while Q ≠ ∅ do
10:        τ ← any element in Q
11:        Q ← Q \ {τ}
12:        hgt[τ] ← τ.t + max_{τ'∈succ[τ]}{hgt[τ']}
13:        vol[τ] ← τ.t + Σ_{τ'∈succ[τ]}{vol[τ']}
14:        for each τ' ∈ pred[τ] do
15:            outdg[τ'] ← outdg[τ'] − 1
16:            if outdg[τ'] = 0 then Q ← Q ∪ {τ'}
17:            end if
18:        end for
19:    end while
20:    for each τ ∈ V do
21:        pri[τ] ← (−hgt[τ], −vol[τ], −|succ[τ]|, τ.id)
22:    end for
23:    return(pri, indg, succ)
24: end function
```

Algorithm 6 ScheduleOBS().

```
1: function ScheduleOBS(T, p, pri, indg, succ)
2:     now ← 0
3:     free ← [1..p]                          ▷ Free cores
4:     running ← ∅                ▷ Trans. that are running
5:     ready ← {τ ∈ T : indg[τ] = 0}          ▷ Ready tx
6:     repeat
7:         while (ready ≠ ∅) ∧ (free ≠ ∅) do
8:             τ ← arg min{pri[τ'] : τ' ∈ ready}
9:             ready ← ready \ {τ}
10:            running ← running ∪ {τ}
11:            start[τ] ← now
12:            end[τ] ← now + τ.t
13:            core[τ] ← any element in free
14:            free ← free \ {core[τ]}
15:        end while
16:        now ← min{end[τ] : τ ∈ running}
17:        S ← {τ ∈ running : end[τ] = now}
18:        running ← running \ S
19:        for each τ ∈ S do
20:            for each τ' ∈ succ[τ] do
21:                indg[τ'] ← indg[τ'] − 1
22:                if indg[τ'] = 0 then
23:                    ready ← ready ∪ {τ'}
24:                end if
25:            end for
26:            free ← free ∪ {core[τ]}
27:        end for
28:    until (running = ∅) ∧ (ready = ∅)
29:    return (start, core)
30: end function
```

Algorithm 7 OBS Heuristics.

Input: Block $T = (\tau_1, \ldots, \tau_n)$. Number of cores p.
Output: A schedule with the start time $start[\tau]$ and core $core[\tau]$ assigned to each transaction $\tau \in T$.

```
G ← BuildDAG(T)
(pri, indg, succ) ← ScoringOBS(G)
(start, core) ← ScheduleOBS(T, p, pri, indg, succ)
```