Omakase: Execution Congestion Control through Conflict-Graph Integration in Multi-Proposer Blockchain

Dakai Kang*, Ray Neiheiser†, Alberto Sonnino‡, Eleftherios Kokoris Kogias‡, Mohammad Sadoghi§
*Sei Labs †Nanyang Technological University, Singapore ‡Mysten Labs
§Exploratory Systems Lab, University of California, Davis, USA
dakai@seinetwork.io, ray.neiheiser@ntu.edu.sg, {alberto, lefteris}@mystenlabs.com, msadoghi@ucdavis.edu

***Abstract*—As multi-proposer protocols push consensus throughput, transaction execution has become the dominant bottleneck in blockchains. Layer-1 blockchains parallelize execution either *pessimistically* (Solana, Sui), requiring upfront state-access declarations and serializing contended transactions, or *optimistically* (Aptos, Sei, Monad), resolving dependencies at run time but suffering cascading aborts under contention. Both treat blocks in isolation, leaving inter-block parallelism unexploited and most cores idle.**

We introduce Omakase, a parallel execution framework that combines pre-execution, inter-block conflict-graph integration, and graph-aware Optimistic Concurrency Control (OCC). Replicas use idle cores to pre-execute only the blocks they propose, spreading proposer-side prediction work across concurrent proposers. After building per-block conflict graphs and identifying their hot resources, a greedy scheduler integrates graphs across the blocks committed in a round, keeping an integration only when a lightweight balance heuristic predicts a parallelism gain. A graph-aware OCC executor then runs the integrated graph, admitting transactions in dependency order, which eliminates aborts when the predicted graph is exact and reduces re-executions when it is not.

On 15,000 Ethereum mainnet blocks (∼2.4M transactions) and synthetic workloads with configurable contention, Omakase reaches a 2.47× speedup over sequential execution, well above OCC and graph-aware baselines without inter-block integration. Against the strongest alternative, which runs the whole round as one optimistic window, Omakase matches mean throughput but is markedly more stable under high contention, because integration bounds how much optimistic work is in flight: this is the congestion control of our title. Integration is not free, but overlapping it with execution across rounds hides 94% of the measured integration cost in the validator pipeline.

***Index Terms*—blockchain, parallel execution, optimistic concurrency control, conflict graphs, congestion control**

I. Introduction

Modern blockchains have made dramatic progress in scaling consensus throughput. *Multi-proposer* protocols enable concurrent block proposals and commit batches of blocks per round. This architecture spans DAG-based protocols [1], [2], [3], [4], [5], [6], [7], [8], [9] (e.g., Mysticeti [10], Bullshark [3]), concurrent consensus instances [11], [12], [13], [14], [15] (e.g., RCC [11]), and multi-proposer dissemination layers [16], [2], [17]. However, once consensus delivers a totally ordered batch of blocks, replicas must still execute the transactions against a shared state. As consensus throughput increases and latency drops, transaction execution has emerged as the dominant bottleneck in end-to-end performance.

To address this bottleneck, high-performance Layer-1 blockchains adopt one of two parallel execution architectures. The *pessimistic* approach forces transactions to declare state accesses upfront, allowing the scheduler to safely partition work across cores. Solana's Sealevel [18], [19] requires account-level declarations, while Sui's Move runtime [20], [21] relies on object-level granularity. Conversely, the *optimistic* approach speculates and validates. Systems like Aptos [22], Sei [23], and Monad [24] deploy Block-STM [25], an *Optimistic Concurrency Control (OCC)* engine. Block-STM executes transactions concurrently against a multi-versioned state, relying on a sequential validator to abort and re-execute transactions whose read sets are invalidated by earlier writers.

Both models present fundamental trade-offs. The pessimistic approach requires runtimes tailored for explicit state declarations. This renders it incompatible with the Ethereum Virtual Machine (EVM), where storage accesses depend on runtime values and cannot be declared in advance. Furthermore, pessimistic execution imposes a restrictive programming model; Sui, for instance, had to add *dynamic objects* discovered at runtime, which the scheduler can only execute sequentially. The optimistic approach is more developer-friendly by avoiding upfront declarations. However, while OCC performs better under low contention, its throughput degrades sharply under heavy contention due to frequent aborts and re-executions.

Moreover, although these engines already exploit *intra-block* parallelism, they still treat each block in isolation, leaving *inter-block* parallelism untapped. Consequently, despite validator hardware commonly having 16 or more cores, practical blockchain workloads limit parallel speedup to only a few cores, leaving most CPU resources idle [26]. Multi-proposer consensus protocols uniquely alter this landscape: because each round commits a batch of blocks concurrently, they naturally expose both intra-block and inter-block parallelism. Crucially, in a multi-proposer setting, the computational cost of pre-executing a block can be cleanly amortized across the network proposers. §IV turns these observations into a design.

Building on these observations, we introduce Omakase, a parallel execution framework that combines pre-execution,

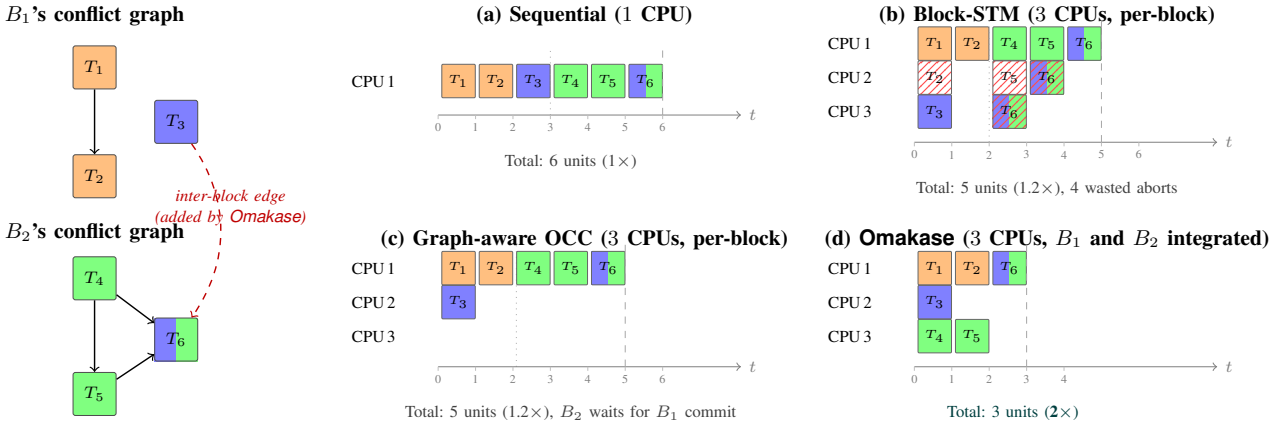


Fig. 1: A motivating example. Two blocks B_1=$\{T_1, T_2, T_3\}$ and B_2=$\{T_4, T_5, T_6\}$ are committed in one round; their per-block conflict graphs are colour-coded by dependency chain, with T_6 split blue/green to mark its inter-block conflict with T_3. We show four execution modes on a 3-CPU machine, with total time units labelled per panel.

inter-block conflict-graph integration, and a graph-aware OCC engine. Replicas leverage idle cores to pre-execute only the blocks they propose: before broadcasting a block, the proposer executes it to predict state accesses and construct a conflict graph. A gas-weighted simulation then reorders the block's transactions so that long dependency chains start early while every predicted conflict keeps its block order, estimates the resulting per-thread load, and identifies hot resources, whose access span dictates the critical path. Upon committing a batch of blocks in a round, a greedy scheduler selectively integrates their conflict graphs, retaining an integration only when a lightweight heuristic predicts a parallelism gain, and re-simulates each integrated graph to fix its execution order. Finally, rather than executing every transaction optimistically as in Block-STM, our graph-aware OCC executor admits a transaction only after the transactions it reads from, according to the conflict graph, have executed. A sequential validation phase then catches any dependency the graph missed.

The title names what integration buys. Running a whole round as one optimistic window is like a network without congestion control: fast while conflicts are rare, collapsing when they are not, because every conflict between blocks becomes a re-execution that can spread through the window. Omakase bounds the window: it considers integrating a block to a group only when its hot-resource set is disjoint from that of the current integrated graph, and accepts the integration only if the balance test passes, so the amount of optimistic work in flight is decided from the graphs before execution, not discovered through aborts during it. The price is some average throughput; the return is a round time that stays close to the typical one even on a bad round.

We implement Omakase on top of Pevm [27] and evaluate it on 15,000 Ethereum mainnet blocks (about 2.4 M transactions) and on synthetic workloads with configurable contention (§VI). On real Ethereum Omakase executes a round 2.47× faster than sequential execution, against 1.89× for Block-STM and 1.66× for the same graph-aware engine run block by block. Running the whole round as one optimistic window is about as fast on average (2.40×) but much less steady: its slowest round runs at 1.51× against 1.86× for Omakase. The integration step costs more than it saves if a validator waits for it, but overlapping it with the execution of the previous round hides 94% of that cost, for a validator-side speedup of 2.23× against 1.91× for Block-STM.

Contributions. This paper makes the following contributions:

- A pre-execution + conflict-graph framework that exposes both intra-block and inter-block parallelism.
- A gas-weighted simulation that reorders a block's transactions along their dependency chains, estimates per-thread load, and identifies hot resources.
- A *formulation* of inter-block graph integration as a balance-constrained scheduling problem, deciding which blocks' conflict graphs to integrate in a round so as to maximize thread utilization.
- A greedy heuristic for grouping conflict graphs using simulated load balance and hot-resource information.
- A congestion-control view of optimistic execution across blocks: bounding the optimistic window keeps the round time stable where one large window collapses under inter-block conflict.
- An open-source implementation [28] on top of Pevm and an evaluation on real and synthetic Ethereum workloads against sequential execution, Block-STM, graph-aware OCC, Vegeta and the concatenated round, showing that Omakase outperforms every per-block engine and matches the concatenated round's throughput while remaining far more stable under contention, together with an accounting of what every stage costs and how overlapping integration with execution hides most of the integration cost in the validator pipeline.

II. Background and Related Work

Parallel execution is a widely explored topic in database research [29], [30], [31], [32] and, more recently, in

blockchains [33]. However, in the blockchain context the result of execution must be deterministic to guarantee that all validators converge on the same state, otherwise safety is violated. This narrows the relevant database literature to deterministic parallel execution systems such as Calvin and Bohm [34], [35], [36], [37], [38], [39]. These engines, however, assume each transaction's read and write set is known before execution. This assumption does not hold for blockchain workloads such as EVM workloads: under the EVM, storage accesses depend on runtime state, so read/write sets cannot be fully known in advance and can at best be predicted.

Moreover, blockchain workloads are characterized by high contention and write-heavy access patterns [26]. This has motivated a body of blockchain-native work. BlockSTM [25], deployed in production by Aptos [22], Sei [23], and Monad [24], is the most prominent example, executing transactions fully optimistically and resolving conflicts at runtime via a multi-version data store. Optimistic approaches, however, underperform under contention: because dependencies are unknown, transactions are executed optimistically and then validated in index order, so a transaction that read a value a lower-indexed transaction later overwrites is aborted and re-executed. Under heavy contention these aborts cascade, and at the limit throughput can fall below sequential execution.

To avoid the re-execution overhead that optimistic approaches incur under contention, a range of systems expose conflict information ahead of execution. One approach places this burden on the client: Solana [18] and Sui [40] mandate that each transaction declares all state it will read or write. These declarations are enforced as correctness conditions rather than treated as estimates, so a transaction that touches an undeclared location fails, while one that over-declares its footprint needlessly constrains the scheduler.

A different approach shifts hint generation to the block proposer [41], [42], [43], [44] where the leader executes transactions at block creation time to estimate the dependencies they will have during run-time. This is often done on the critical path of consensus [41], which results in deterministic hints but caps the throughput to the pre-execution speed. To avoid this, a recent approach [45] removes the dependency on the proposer altogether by having each node independently infer conflict sets through static analysis of EVM and MoveVM bytecode before execution begins. Nevertheless, this analysis also needs to run on the critical path which significantly constrains the execution speed-up.

Multi-proposer consensus protocols [10], [13], [14], in which several nodes concurrently produce blocks each round, have grown in prominence, yet parallel execution has rarely been studied in this setting. Vegeta [46] is the closest existing approach aimed at improving execution throughput in blockchains with multiple concurrent proposers. It proposes to spread the pre-execution workload across concurrent leaders to relieve the single-proposer bottleneck. Even so, multiple proposers in Vegeta serve only to pipeline hint generation for what ultimately becomes one block. Thunderbolt [47] pre-executes transactions *before* ordering: a transaction whose pre-executed read set fails validation causes its block to be discarded. We reimplement Vegeta and compare against it in §VI-B; Thunderbolt's block-discarding failure model has no counterpart in our setting, where every committed transaction must execute, so we position it analytically only.

However, the parallelism that naturally arises from executing transactions across independently proposed blocks simultaneously is still left unexploited. Omakase pursues precisely this opportunity: by treating concurrently proposed blocks as independent execution streams whose non-conflicting transactions can proceed in parallel, we maximize execution parallelism.

Work such as Anthemius [48] that filter block content to maximize throughput is orthogonal and would further increase the cross-block parallelism potential that Omakase can leverage.

III. System Model

We consider a multi-proposer blockchain with N processes $p_1, \ldots, p_N$, of which up to f may be Byzantine [49]. The consensus protocol Π is a black box: the proposers input blocks $B_1, \ldots, B_N$ and Π outputs the same ordered sequence of blocks at every correct process. Every correct process then executes the committed transactions, which must be deterministic so that all processes reach the same state; as in Ethereum, Aptos and Sui [50], [22], [51], execution cost is measured in gas. Unlike prior work [48], [52], [53], we require neither clients nor the system to declare transaction dependencies or gas upfront.

Transactions, resources, and gas. A *transaction* T_i is a deterministic program. Run against a state σ, it reads a set $R_i(\sigma)$ of *resources* (storage slots and account state), writes a set $W_i(\sigma)$, and consumes gas $g_i(\sigma)$. The sets are state-dependent and therefore cannot be known before execution; Omakase predicts them by pre-executing the block and treats the resulting conflict graph only as a scheduling hint (§V-A). Two transactions *conflict* on a resource when both access it and at least one writes it.

Concurrent execution. An *incarnation* (i, c) is the c-th execution attempt of T_i; a new incarnation starts whenever the previous one fails validation. In a concurrent execution of a round $T_1, \ldots, T_n$ on k threads, incarnations run in parallel against a *multi-version* state: an incarnation of T_i reads each resource from the highest-indexed writer below i (or the pre-round state) and writes versions tagged (i, c). Validation then re-reads, in index order, every resource that T_i's latest incarnation read; if any value has since been overwritten by a lower-indexed transaction, T_i is re-executed (§V-F). The execution *commits* once every transaction has passed validation.

Serializability. The committed state must equal that of executing $T_1, \ldots, T_n$ one by one in a single fixed order, which Omakase derives deterministically from the round (§V-B, §V-E), so every correct replica converges on the same state. The order keeps every predicted conflict as the blocks have it (§V-B).

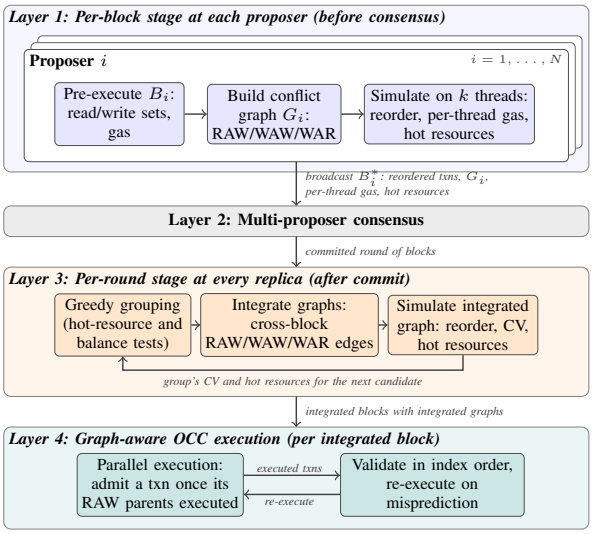


Fig. 2: End-to-end architecture of Omakase. **Layer 1**: each of the N proposers pre-executes the block it proposes, builds its conflict graph, simulates it on k threads to reorder the transactions, estimate per-thread gas and identify hot resources, and broadcasts all of this with the block. **Layer 2**: consensus commits a round of blocks. **Layer 3**: every replica greedily integrates the committed graphs and re-simulates each integrated graph. **Layer 4**: each integrated block runs on the graph-aware OCC engine, whose sequential validation catches mispredictions.

IV. System Overview

Omakase runs at every replica and slots into a multi-proposer consensus protocol at two points: when a replica proposes a block, and when a round of consensus commits a batch of blocks. It targets two opportunities that current engines leave unused. *Within* a block, OCC discovers conflicts by aborting; a conflict graph predicted before execution lets the engine schedule around most of them instead. *Across* blocks, engines execute one block at a time even when consecutive blocks touch disjoint hot resources; the same predicted graphs make it possible to integrate such blocks and run their transactions side by side.

Two observations make this practical.

1) *Prediction is affordable.* With balanced proposal rates and comparable block costs, each replica performs proposer-side prediction work for approximately $1/N$ of the committed blocks. This amortizes computational work across proposers.
2) *The conflict graphs carry enough information to decide integration.* A block's graph, annotated with predicted gas, summarizes its dependencies: graphs of different blocks can be integrated without executing anything, their hot resources tell which pairs would only lengthen the critical path, and a single-threaded simulation that walks the integrated graph on k threads estimates per-thread load at a fraction of execution cost. Together, these summaries support a lightweight heuristic that selects integrations expected to improve worker utilization.

Figure 2 shows how the two stages realize them.

Per-block stage (before consensus). When a replica is about to broadcast a block, it pre-executes the block's transactions against the latest committed state to predict each transaction's read set, write set and gas. From these predictions it builds the block's *conflict graph*, whose nodes are transactions and whose edges record every conflict on a shared resource: a read after a write, two writes, or a write after a read (§V-A). A single-threaded scheduler then simulates parallel execution on k threads to reorder the transactions, keeping every predicted conflicting pair in block order, to estimate per-thread load (§V-B), and to identify *hot resources*, whose access span dominates (§V-C). The graph, the reordered transaction list, the per-thread load and the hot-resource set are broadcast with the block, so the whole stage is paid once, by the proposer.

Per-round stage (after consensus commits). When a round commits a batch of blocks, every replica already holds each block's conflict graph and hot-resource set. A greedy scheduler considers a block for integration to a group only when its hot-resource set is disjoint from that of the group's current integrated graph, and accepts the integration only if a balance test passes (§V-D, §V-E). Each integrated graph is re-simulated to fix the execution order of its transactions, exactly as a single block is (§V-B), and then executed by a graph-aware OCC engine that runs transactions in parallel against a multi-version state and admits a transaction only after the transactions it reads from have executed. Then a validation phase in transaction-index order catches every misprediction (§V-F). Groups are executed one after another, so a conflict between two groups never goes through validation.

Why balanced integrations? Integration is constrained by the replica's hardware along two dimensions. The single-core execution speed bounds the gas-weighted *depth* of a conflict graph: each chain of dependent transactions must execute sequentially, though not necessarily on the same core, so its gas is a lower bound on execution time. The core count k bounds the *width*: how many chains can proceed side by side. An integration is therefore expected to pay off only when the integrated graphs fill the cores that are idle otherwise, without making the critical path that a single core must execute longer. This is the congestion control of our title: the optimistic window should grow only when it fills idle cores.

V. The Omakase Framework

A. Pre-Execution & Conflict Graph Construction

When proposing a block, a replica pre-executes its transactions to predict each one's read set, write set and gas (Algorithm 1, Line 6)[1], and builds the block's conflict graph by inserting the transactions in the order they appear in the block (Line 5). Fig. 3 traces this and the two algorithms of §V-B on one block of 6 transactions.

[1]We assume that no transaction fails due to nonce mismatch.

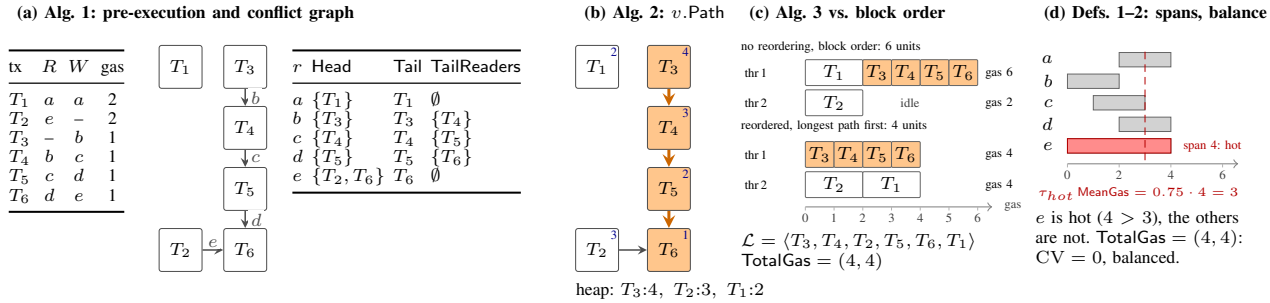


Fig. 3: Algorithms 1–3 and the hot-resource test on a six-transaction block, with k=2 worker threads. (a) Predicted accesses, the conflict graph and the per-resource maps. (b) v.Path, the longest path starting from each transaction node v, with the critical path highlighted. (c) In block order the critical path starts at offset 2 and runs alone ($\mathsf{TotalGas} = (6, 2)$); Algorithm 3 starts it first ($\mathsf{TotalGas} = (4, 4)$) and yields $\mathcal{L}$. (d) Access spans in gas offsets: with τ_{hot}=0.75, e exceeds $\tau_{hot} \cdot \mathsf{MeanGas} = 3$ and is hot. The $\mathsf{TotalGas} = (4, 4)$ gives this conflict graph CV $= 0$.

The graph records every conflict between two transactions of the block. Each node is a transaction annotated with:

- **Read set, write set, gas**: the predicted accesses and cost. The read set covers storage slots and account state (balance, nonce, code) alike, excluding only the block beneficiary.
- **Parents and children**: an edge $u \to v$ means that u precedes v in the block and the two conflict on a resource: v reads what u wrote (RAW), overwrites it (WAW), or writes what u read (WAR); u is a parent of v and v a child of u.

For each resource r the construction maintains:

- **Tail**: the latest writer of r. A new reader of r depends only on the tail, since an edge to the last writer subsumes all earlier accesses (Lines 9–12); a new writer of r depends on the tail (WAW, Line 14) and becomes the tail (Lines 13–17).
- **Tail readers**: the readers of r since its last write. They are the WAR parents of r's next writer, which clears the set (Line 16).
- **Heads**: the transactions that access r before its first write within the block, that is, r's entry points with no incoming dependency on r. They are where another block's tail attaches to this graph when graphs are integrated (§V-E).

The graph is a *scheduling hint*, not the serialization mechanism. It may be imprecise because a transaction's accesses depend on the state it executes against: pre-execution runs the block against the state at proposal time, whereas the committed execution runs it after the preceding transactions of the round, including those of other blocks. If the state has changed, the transaction may touch resources its prediction did not include. Correctness never depends on the graph being exact (§V-F).

Algorithm 1: Pre-execution and Conflict Graph Construction

Input: Ordered block transactions $\mathcal{T} = \langle tx_1, \ldots, tx_n \rangle$
Output: Conflict graph $G = (V, E)$; per-resource maps Head, Tail, TailReaders

1 $G \leftarrow (\emptyset, \emptyset)$
2 Head $\leftarrow \emptyset$ `// resource ↦ head nodes (default ∅)`
3 Tail $\leftarrow \emptyset$ `// resource ↦ last writer (default ⊥)`
4 TailReaders $\leftarrow \emptyset$ `// resource ↦ readers since its last write`
5 **for** $i \leftarrow 1$ **to** n **do**
6 $(R_i, W_i, g_i) \leftarrow$ SeqExecPredict(tx_i) `// seq. pre-exec of` tx_i`: predict` R, W`, gas`
7 $v_i \leftarrow$ AddNode(G, tx_i); $v_i.R \leftarrow R_i$; $v_i.W \leftarrow W_i$; $v_i.gas \leftarrow g_i$
8 $P \leftarrow \emptyset$ `// parents of` v_i
`// Reads: RAW edge from the last writer, else a new head`
9 **foreach** $r \in R_i$ **do**
10 **if** Tail$[r] \neq \bot$ **then** $P \leftarrow P \cup \{\mathsf{Tail}[r]\}$
11 **else** Head$[r] \leftarrow$ Head$[r] \cup \{v_i\}$
12 TailReaders$[r] \leftarrow$ TailReaders$[r] \cup \{v_i\}$
`// Writes: WAW edge from the last writer, WAR edges from its readers`
13 **foreach** $r \in W_i$ **do**
14 **if** Tail$[r] \neq \bot$ **then** $P \leftarrow P \cup \{\mathsf{Tail}[r]\}$
15 **else** Head$[r] \leftarrow$ Head$[r] \cup \{v_i\}$
16 $P \leftarrow P \cup$ TailReaders$[r]$
17 Tail$[r] \leftarrow v_i$; TailReaders$[r] \leftarrow \emptyset$
18 **foreach** $u \in P \setminus \{v_i\}$ **do**
19 $E \leftarrow E \cup \{(u, v_i)\}$; u.Children $\leftarrow u$.Children $\cup \{v_i\}$; v_i.Parents $\leftarrow v_i$.Parents $\cup \{u\}$
20 **return** $(G,$ Head, Tail, TailReaders$)$

B. Reordering Transactions via Simulation

Consider the block of Fig. 3 on $k = 2$ threads. Scheduled in the order the transactions appear (Fig. 3c, top), the two independent transactions go first, and the chain $T_3 \to T_4 \to T_5 \to T_6$ starts only at offset 2 and then runs alone for four units while the other thread idles: the threads carry 6 and 2 gas and the block takes 6 units. Starting the chain first (bottom) lets T_1 and T_2 fill the other thread: 4 and 4 gas, done at 4.

Omakase reorders the block's transactions by simulating their parallel execution on k threads (Algorithm 2), scheduling the transaction with the most expensive remaining chain first to shorten the critical path.

Concretely, it first computes for every node the cost of the most expensive path starting there (Lines 1–6): gas is attached to nodes, so a path's cost is the sum of the gas of the transactions on it, and a node's value is its own gas plus the maximum value of its children, computed by depth-first search with memoization (Fig. 3b: T_3 has 4, and T_2 has 3

Algorithm 2: TxnReorder

```
Input: Conflict graph G = (V, E); number of threads k
Output: Reordered list of transaction nodes L
// (1) Compute longest gas-weighted path from
   each node (DFS + memoization)
1  foreach v ∈ V do
2      v.RemParents ← | v.Parents |
3      v.Path ← ⊥
4  foreach v ∈ V do
5      if v.Path = ⊥ then
6          DFSPath(v)
// (2) Initialize max-heap keyed by v.Path with
   all nodes without parents
7  H ← MaxHeap()
8  foreach v ∈ V do
9      if v.RemParents = 0 then
10         Push(H, v, key = v.Path)
// (3) Simulate k-thread execution and output
   the reordered list
11 (L, TotalGas) ← SimReorder(G, H, k)
12 return (L, TotalGas)
```

Algorithm 3: SimReorder: Single-Thread Simulation of k Parallel Threads

```
Input: Graph G = (V, E); max-heap H; number of threads k
Output: Reordered list L; per-thread total execution time TotalGas
1  L ← []
2  for t ← 1 to k do
3      Offset[t] ← 0          // gas offset when t is free
4      Assigned[t] ← ⊥                  // assigned node
5      TotalGas[t] ← 0  // total gas executed by thread
         t
6  while |L| < |V| do
       // Assign ready nodes to idle threads
          (smallest idle thread id)
7      while ∃t : Assigned[t] = ⊥ ∧ ¬Empty(H) do
8          t0 ← min{t | Assigned[t] = ⊥}
9          v ← PopMax(H)
10         Assigned[t0] ← v
11         v.start ← Offset[t0]   // gas offset at which v
             begins
12         Offset[t0] ← Offset[t0] + v.gas
       // Select earliest finishing active thread
13     t* ← arg min_{t: Assigned[t]≠⊥} Offset[t]
14     g_min ← Offset[t*]
       // Synchronize idle threads
15     for t ← 1 to k do
16         if Assigned[t] = ⊥ then
17             Offset[t] ← g_min
       // Mark node as executed
18     v* ← Assigned[t*]
19     Assigned[t*] ← ⊥
20     TotalGas[t*] ← TotalGas[t*] + v*.gas    // accumulate
         execution cost
21     Append(L, v*)
       // Release children
22     foreach c ∈ v*.Children do
23         c.RemParents ← c.RemParents − 1
24         if c.RemParents = 0 then
25             Push(H, c, key = c.Path)
26 return (L, TotalGas)
```

through its WAR edge to T_6).

It then runs the simulation (Line 11, Algorithm 3; Fig. 3c, bottom): transactions whose parents have all been scheduled are pushed into a max-heap keyed by path cost (Lines 7–10), initially T_3, T_2 and T_1. Then an idle simulated thread takes the top of the heap and advances by that transaction's gas (Algorithm 3, Lines 7–12), and a transaction joins the reordered list $\mathcal{L}$ when its simulated thread finishes it (Lines 18–21), giving $\mathcal{L} = \langle T_3, T_4, T_2, T_5, T_6, T_1 \rangle$ in Fig. 3.

What the new order preserves. $\mathcal{L}$ becomes the order in which the block's transactions are serialized (§III). Since every predicted conflict is an edge, $\mathcal{L}$, a topological order of the graph, keeps each such pair as the block has it. Validation (§V-F) then guarantees that the committed state is that of executing the transactions one by one in $\mathcal{L}$ order. The graph does not change with the order, only the transaction indices do; the schedule the simulation computed, with its per-thread gas and gas offsets, therefore describes the reordered block and feeds §V-C and §V-D.

C. Identifying Hot Resources

Two conflict graphs should not be integrated when their transactions contend on the same resource for much of the block: the conflicting chains cannot run in parallel, so integration would only lengthen the critical path. We therefore identify, in each graph, the resources that stay in active use for a large part of the block. Execution exposes no wall-clock time at proposal time, so "in use" is measured in *gas*, the deterministic cost of each transaction: the simulation of §V-B tracks a gas offset per thread ($\mathsf{Offset}[t]$ in Algorithm 3), and records for every transaction the offset of its thread when the transaction is assigned (Line 11).

Definition 1 (Resource Access Span)**.** *For each transaction tx, let $g_{tx}^{start} = tx.\mathsf{start}$ be the gas offset recorded by Algorithm 3, and $g_{tx}^{end} = g_{tx}^{start} + \mathrm{gas}(tx)$. For a resource r, let $\mathcal{T}_r$ be the set of transactions that access r, and define*

- $g_r^{first,start} = \min_{tx \in \mathcal{T}_r} g_{tx}^{start}$*, the gas offset at which the first access to r begins,*
- $g_r^{last,end} = \max_{tx \in \mathcal{T}_r} g_{tx}^{end}$*, the gas offset at which the last access to r ends.*

The difference $g_r^{last,end} - g_r^{first,start}$ is the $r.\mathsf{span}$ of resource r.

Let $\mathsf{MeanGas}$ be the average total gas of the k simulated threads. A resource is *hot* if $r.\mathsf{span} > \mathsf{MeanGas} \cdot \tau_{hot}$, where τ_{hot} is a configurable threshold. Fig. 3(d) applies the test to the example block with τ_{hot}=0.75: e, accessed at both ends of the block, has span 4 against a threshold of 3 and is hot, while the other resources have span 2 and are not hot.

D. Identifying Balanced Conflict Graphs

A conflict graph whose dependency chains leave simulated threads idle is a candidate for integration, because its idle capacity can absorb another graph's dependency chains. An integration is worthwhile only if it fills that capacity instead of making the critical path relatively longer.

In Fig. 3(c), block order gives the two threads 6 and 2 gas in total: the dependency chain occupies one thread while the other is idle for four units, and that idle capacity is exactly what another graph's dependency chains could fill. The reordered schedule gives them 4 and 4: nothing is idle, so integrating more would only make the critical path relatively longer.

We therefore measure balance by the coefficient of variation of the per-thread gas: with $\mathsf{TotalGas}[t]$ the gas of thread t, μ and σ its mean and standard deviation over the k threads, and $\mathrm{CV} = \sigma/\mu$, $\mathsf{TotalGas} = (6, 2)$ gives $\mathrm{CV} = 0.5$ and $(4, 4)$ gives $\mathrm{CV} = 0$.

Definition 2. *Given a simulation with k threads and a threshold τ_{CV}, a conflict graph is* (k, τ_{CV})-balanced *if and only if* $\mathrm{CV} \le \tau_{\mathrm{CV}}$.

An integration of graphs $G_1, \ldots, G_m$ (§V-E) is judged against its parts: it must be at least as balanced as they were on average.

Definition 3. *An integrated conflict graph G of $G_1, \ldots, G_m$ is a* (k, τ_{CV})-balanced integration *if and only if*

$$\mathrm{CV}_G \le \mathrm{CV}_{\mathrm{base}} = \sum_{i=1}^{m} w_i \, \mathrm{CV}_{G_i}, \qquad w_i = \frac{\mu_i}{\sum_{j=1}^{m} \mu_j}.$$

This condition compares the dispersion of simulated per-thread gas loads. We use it as a proxy for the utilization benefit of integration.

E. Integrating Conflict Graphs

Integration decides which of a round's graphs to integrate into one group and how to integrate them. A greedy pass over the graphs (Algorithm 4) makes the first decision with the balance test of §V-D and the hot resources of §V-C; a pairwise integration (Algorithm 5) performs the second.

Choosing the groups. Algorithm 4 sorts the graphs by descending CV (Line 1), seeds a group with the highest-CV graph (Line 4), and tries the remaining graphs from the lowest CV upward (Line 6). For each candidate block B, the integrator compares $\mathrm{Hot}(B)$ with $\mathrm{Hot}(G)$, where G is the current integrated graph, and rejects the candidate when these sets intersect. Otherwise it tentatively integrates B into G, simulates the result, and applies the balance test (Lines 7–8). For an accepted integration, the simulated hot-resource set becomes $\mathrm{Hot}(G)$ for the next decision. The traversal stops as soon as the group is itself balanced (Line 11), or when the group reaches a cap of 10 blocks, a safeguard that rarely binds (§VI-A). The integrated graphs leave the list (Line 13) and the procedure repeats until every graph is grouped. The test uses the integrated graph at the time of each decision. Because its hot-resource set is recomputed after integration, it does not guarantee pairwise disjointness among the original per-block hot-resource sets in the final group.

Why this greedy. The highest-CV graph seeds a group because it has the most idle capacity to fill; low-CV candidates come first because they consist of many short chains that slot into idle threads. A candidate that shares a hot resource with the current integrated graph is never integrated, since their chains on that resource would run one after the other anyway. An integration is kept only if the group is at least as balanced as its parts ($\mathrm{CV}_G \le \mathrm{CV}_{\mathrm{base}}$). Once the group is balanced ($\mathrm{CV}_G \le \tau_{\mathrm{CV}}$) integration stops: adding more is expected to bring little further gain while widening the optimistic window and risking more re-executions.

Algorithm 4: GREEDYBALANCEDINTEGRATION

Input: A set of conflict graphs $\{G_1, G_2, \ldots, G_m\}$
Output: A list of integrated graphs $\mathcal{O}$

```
1  Sort the graphs in descending order of CV and store them in a list L
2  O ← []
3  while L ≠ ∅ do
4      G ← L[1]  // Initialize with highest-CV graph
5      S ← {L[1]}          // Graphs integrated into G
6      for i ← |L| to 2 do
7          if BalancedIntegration(G, L[i]) then
8              G ← Integrate(G, L[i])
9              S ← S ∪ {L[i]}
10         if BalancedGraph(G) then
11             break
12     Append G to O
13     Remove all graphs in S from L
14 return O
```

Fig. 4 runs the greedy on a three-block round.

Integrating two graphs. Integrating G_2 into G_1 treats G_2's block as if it followed G_1's: every cross-block conflict becomes an edge from the G_1 transaction to the G_2 transaction, and pairs that do not conflict get no edge and may run in parallel. The per-resource maps make this a local operation: for a resource r, only G_1's last writer and the readers since that write can conflict with G_2, and only G_2's heads (its accessors before its first write of r) and that first writer can conflict with G_1. In Fig. 4, integrating B_2 after B_1 adds the single edge $T_1 \to T_6$: T_1 is G_1's last writer of y and T_6 is G_2's only accessor of it.

Algorithm 5 connects exactly those pairs, resource by resource (Line 3): G_1's tail to every head of r in G_2, a RAW edge to each reader before G_2's first write and a WAW edge to that writer (Lines 5–6), and G_1's tail readers to G_2's first writer (WAR, Line 11). The maps are then integrated so that G can be integrated further: the heads of r stay G_1's where G_1 wrote r and gain G_2's otherwise (Line 8); the tail and tail readers become G_2's where G_2 wrote r (Line 13), and G_2's readers join G_1's tail readers otherwise (Line 15).

Finally the integrated graph is simulated once more, exactly as a single block's graph is (§V-B, Line 17). The simulation recomputes the transaction order, CV and hot resources in the integrated graph.

F. Graph-Aware OCC Engine

To enable parallel execution, we deploy a graph-aware OCC engine (Algorithm 6), a variant of Block-STM that follows the conflict graph during execution. Transaction indices are

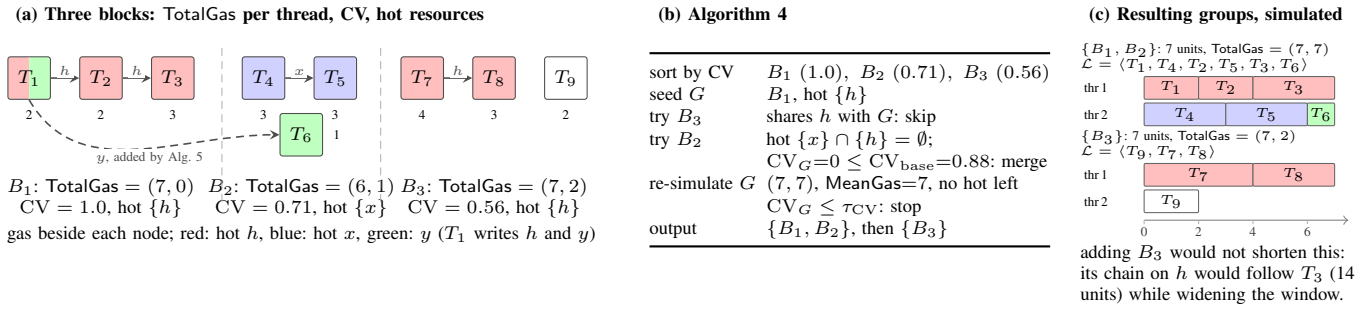


Fig. 4: Integration of a three-block round (k=2, τ_{hot}=1.5, τ_{CV}=0.5). (a) Per-block simulation: B_1 and B_3 each has a hot resource h, B_2 has a hot resource x. (b) The greedy integration starts with B_1, skips B_3 because it shares the hot resource h, and integrates B_2, whose hot resource differs. Algorithm 5 adds the dashed edge $T_1 \to T_6$. (c) The two groups take 14 units in sequence against 20 for the three blocks alone; adding B_3 to the group would serialize both chains on h and gain nothing.

Algorithm 5: IntegrateGraphsAndReorder: Integrate G_1 and G_2 and Reorder Transactions

Input: $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, each with its maps Head, Tail, TailReaders (a resource a graph never touched reads as ∅, ⊥, ∅); number of threads k
Output: Reordered list of transactions $\mathcal{L}$

1 $V \leftarrow V_1 \cup V_2$; $E \leftarrow E_1 \cup E_2$
2 (Head, Tail, TailReaders) $\leftarrow$ (G_1.Head, G_1.Tail, G_1.TailReaders)
// Connect every cross-graph conflict on each resource G_2 accesses
3 **foreach** r accessed in G_2 **do**
4 **if** Tail[r] $\neq \perp$ **then**
// G_1 wrote r: RAW edges to G_2's readers before its first write, WAW edge to that writer
5 **foreach** $v \in G_2$.Head[r] **do**
6 AddEdge(E, Tail[r], v)
7 **else**
8 Head[r] $\leftarrow$ Head[r] $\cup$ G_2.Head[r]
9 **if** G_2.Tail[r] $\neq \perp$ **then**
// G_2 wrote r: WAR edges from G_1's readers since its last write to G_2's first writer
10 $w \leftarrow$ the node of G_2.Head[r] that writes r
11 **foreach** $u \in$ TailReaders[r] **do** AddEdge(E, u, w)
12 Tail[r] $\leftarrow$ G_2.Tail[r]
13 TailReaders[r] $\leftarrow$ G_2.TailReaders[r]
14 **else**
15 TailReaders[r] $\leftarrow$ TailReaders[r] $\cup$ G_2.TailReaders[r]
// After integration, reorder transactions using the transaction reordering algorithm
16 $G \leftarrow$ ((V, E), Head, Tail, TailReaders)
17 ($\mathcal{L}$, TotalGas) $\leftarrow$ TxnReorder(G, k)
18 **return** ($\mathcal{L}$, TotalGas)

positions in the order fixed by the simulation (§V-B, §V-E), so multi-version reads and validation follow the fixed order. Transactions are executed in parallel, and the graph decides when a transaction becomes eligible.

Correctness rests on *sequential validation*, not on the graph. Because the predicted read and write sets may be inaccurate, each executed transaction is validated in increasing index order: validation re-reads every resource the transaction observed from the multi-version state, and if a lower-indexed transaction has since written it, the read is invalid and the transaction is re-executed as a new incarnation (Line 9). Every re-execution, and every incarnation that writes a resource its predecessor did not, puts all higher-indexed transactions back into the validation queue (Lines 9 and 16), so no transaction commits having read a version that a lower-indexed transaction later changed. Validation is prioritized over execution.

We maintain an execution task pool ETask, which contains transactions whose RAW parents have all executed and that are ready for execution (Line 1). After a transaction is executed, a corresponding validation task is created and added to the validation task pool VTask (Line 13). At the same time, any child transactions whose dependencies have now been satisfied are inserted into ETask (Line 14). When selecting tasks from ETask and VTask, we prioritize tasks with lower transaction indices (Line 6).

The key difference from Block-STM is how ETask is maintained: Block-STM starts with every transaction in it, whereas our engine admits only transactions whose RAW parents in G have all executed (Line 1). Only RAW parents are waited for: sequential validation makes the other two kinds of edge unnecessary at execution time. A write by a higher-indexed transaction is never visible to a lower-indexed reader, whose reads resolve to versions below its own index (Line 4). If two writers write different values, the one with higher index wins. And when a re-execution happens, the transaction and all transactions with higher indices should be validated again. A missing RAW parent, by contrast, means a read that validation will fail, so admitting a transaction before its RAW parents have executed should not be allowed.

VI. Evaluation

Our evaluation answers four questions.

1) How much faster does Omakase execute a round than single-block parallel execution engines, and than the inter-block baselines that execute the whole round as one optimistic window (§VI-B)?
2) Under which conflict pattern and parameters does integration pay off (§VI-C)?
3) Why does it work, in terms of aborts (§VI-D)?

Algorithm 6: ParExecOCCDG: Parallel Execution with OCC + Conflict Graph

Input: Transaction list $\mathcal{L} = \langle tx_1, \ldots, tx_n \rangle$; conflict graph G; number of threads k
Output: Final execution results for all transactions

ETask $\leftarrow \{tx_i \mid tx_i$ has no unexecuted RAW parents in $G\}$ // ready-to-execute
VTask $\leftarrow \emptyset$ // executed but not validated
for $i \leftarrow 1$ **to** n **do** Inc$[i] \leftarrow 0$
// Multi-version state used by Execute
Read(tx_i, r) returns the version of r written by the highest-indexed tx_j with $j < i$, or the pre-round value; Write(tx_i, r, v) creates version $(i, \text{Inc}[i])$
while ETask $\neq \emptyset \ \vee$ VTask $\neq \emptyset$ **do**
  (kind, i) $\leftarrow$ PickTask(VTask, ETask) // validation before execution; lowest index first
  if kind $=$ Validate **then**
    if $\neg$Validate$(tx_i,$ Inc$[i])$ **then**
      Inc$[i] \leftarrow$ Inc$[i] + 1$; ETask $\leftarrow$ ETask $\cup \{i\}$ // abort and re-execute
      for $j \leftarrow i + 1$ **to** n **do**
        if $j \notin$ ETask **then** VTask $\leftarrow$ VTask $\cup \{j\}$
  else
    $(R, W) \leftarrow$ Execute$(tx_i,$ Inc$[i])$; VTask $\leftarrow$ VTask $\cup \{i\}$
    foreach $j \in tx_i$.Children **do**
      if RawParentsExecuted(j) **then**
        ETask $\leftarrow$ ETask $\cup \{j\}$
    if WritesNewResource$(i,\ W)$ **then**
      for $j \leftarrow i + 1$ **to** n **do**
        if $j \notin$ ETask **then** VTask $\leftarrow$ VTask $\cup \{j\}$

4) And what does each stage cost, including the stages that single-block engines do not have, and how much of that cost can a validator's pipeline hide (§VI-E)?

A. Methodology

Implementation. Omakase is about 6,500 lines of Rust on top of pevm [27], an open-source Block-STM-style parallel EVM executor. On top of it, we implement conflict-graph construction, the reordering simulation, the greedy integrator and the graph-aware OCC engine.

Testbed. One Amazon EC2 `c4.8xlarge` instance: two Intel Xeon E5-2666 v3 sockets (18 cores, 36 hardware threads, 2.9 GHz), 58 GiB of DRAM, Ubuntu 24.04.

Baselines. All engines share one code base, one `revm` and one in-memory state backend, and every hint-based engine consumes the access sets of the same sequential pre-execution, so no baseline receives better hints than Omakase.

- *Sequential*: single-threaded `revm`.
- *Block-STM*: each block executed with pevm's optimistic multi-version engine.
- *Graph-aware OCC*: each block executed with the engine of §V-F on the block's own conflict graph.
- *Concatenated Block-STM* and *concatenated graph OCC*: all blocks of a round integrated, in order, into one block and run with unmodified Block-STM, or with the graph-aware engine on a conflict graph built over the whole round. These are the trivial inter-block baselines: the largest possible optimistic window, no reordering and no hot-resource isolation.

```
mapping(bytes32 => uint256) public state;

function execute(bytes32[] calldata reads,
                 bytes32[] calldata writes,
                 uint256 target) external {
  uint256 count = 0;
  while (count < target) {
    for (uint i = 0; i < reads.length
                     && count < target; i++) {
      state[reads[i]];       count++;  // load
    }
    for (uint i = 0; i < writes.length
                     && count < target; i++) {
      state[writes[i]] += 1; count++;  // increment
    }
  }
}
```

Fig. 5: The simulator contract behind the synthetic and artificial workloads. A transaction is one call to `execute`: the two key lists fix which slots it touches and `target` fixes how much work it does.

TABLE I: Default configuration.

Parameter	Default
Workers k (all engines)	8
Hot-resource threshold τ_{hot}	1.5
Balance threshold τ_{CV}	0.5
Max blocks per group	10
Number of rounds	100 blocks; 150 rounds

- *Vegeta* [46], following its paper's Algorithms 1–3: transactions are re-sequenced so that the longest per-key dependency chains run first, then partitioned along the speculative dependency DAG into batches whose transactions run in parallel against the state at the start of the batch. A transaction whose actual access set differs from the speculated one is executed sequentially at the end. It has no multi-version store and is intra-block only.
- *Omakase*: the per-block graphs integrated into groups (§V-E), each group executed with the graph-aware OCC engine.

Workloads.

- *Real Ethereum*: 15,000 mainnet blocks in three contiguous 5,000-block segments, 16,774,645–16,779,644 (March 2023), 18,581,726–18,586,725 (November 2023) and 19,557,289–19,562,288 (April 2024), about 2.4 M transactions in all. For every block an archive node supplies the header, the full transaction objects and the prestate value of every account and storage slot the block touches.
  EVM execution is state-dependent: a transaction scheduled ahead of one it followed on mainnet observes a different state, may branch to other keys, and causes aborts that are artifacts of replay rather than of scheduling. To avoid fee failures, account balances are raised so that no reordering fails a fee check. To avoid nonce failures, the engines that reorder renumber each sender's nonces in the executed order. For the same reason this is the workload in which hints are imperfect: pre-execution records each transaction's access sets in mainnet order, and a transaction executed in another order can touch keys its hint did not predict. The next two workloads are not subject to these effects.
- *Synthetic*: a synthetic version of the real blocks that keeps their conflict topology and execution-time distribution but is

TABLE II: Comparison between engines. Throughput, speedup, per-block latency, and re-executions

	Real Ethereum					Synthetic				
Engine	tx/s	speedup	p50 ms	p99 ms	re-exec/tx	tx/s	speedup	p50 ms	p99 ms	re-exec/tx
Sequential	17,322	1.00×	8.91	15.00	–	12,097	1.00×	12.50	22.77	–
Block-STM	32,823	1.89×	4.58	9.76	0.55	24,381	2.02×	6.06	17.04	0.73
Concatenated Block-STM	41,523	2.40×	3.67	8.45	0.88	34,186	2.83×	4.41	13.17	1.16
Concatenated graph OCC	37,501	2.17×	4.05	8.55	0.55	36,181	2.99×	4.18	12.97	0
Graph-aware OCC	28,805	1.66×	5.35	10.45	0.27	25,156	2.08×	5.86	16.80	0
Vegeta	22,966	1.33×	6.81	12.26	–	16,558	1.37×	9.13	20.04	–
Omakase	**42,757**	**2.47**×	**3.73**	**5.50**	0.31	27,627	2.28×	5.36	15.19	0

insensitive to order, in the manner of [26]. We performed a one-time sequential replay of every block against its prestate, recording each transaction's read set and write set (one key for each storage slot or account balance), its wall-clock execution time, and its sender.

One simulator contract (Fig. 5) holds a single `mapping(bytes32 => uint256)` and exposes one function, `execute(reads, writes, target)`, that cycles over the two key lists, one load per read key and one increment per write key, until exactly `target` storage accesses have been performed. In order to make the execution time match those of the real Ethereum transactions, each synthetic transaction calls this function with the recorded key lists and $\texttt{target} = \lfloor t_{\text{exec}}/T_{\text{SLOAD}} \rfloor$, where $T_{\text{SLOAD}} \approx 0.48\,\mu$s is the measured cost of one storage access on our testbed.

Each distinct sender is mapped to one fresh pre-funded account that is reused across all blocks, which can preserve nonce chains across blocks. The keys a transaction touches are fixed by its calldata, so hints are exact.

- *Artificial*: the same simulator contract with generated access sets whose conflict is controlled by two parameters, ρ_{intra} for conflict inside a block and ρ_{inter} for conflict across blocks, so that §VI-C can vary conflict patterns while everything else is fixed. Every transaction writes three cold keys drawn uniformly from a pool of 10,000, and every block is assigned one *hot key* from a set $\{h_1, \ldots, h_H\}$. Because every write is an increment, the transactions that write the same hot key form a serial chain, and the two parameters set the shape of that chain:
  - ρ_{intra} controls intra-block conflict: the first ρ_{intra} percent of a block's transactions also write the block's hot key, and the block is then shuffled so that they are spread across it. It sets how many transactions of one block chain on the same key.
  - ρ_{inter} controls inter-block conflict: consecutive blocks are taken in groups of $H-1+\rho_{\text{inter}}$; the first ρ_{inter} blocks of a group all use h_1 and the remaining $H-1$ blocks use $h_2, \ldots, h_H$, one each. It sets how many consecutive blocks chain on the same key.

  The cold keys add only a random, unstructured background of conflicts; per-transaction work is fixed at 100 storage accesses and every transaction has a fresh sender, so the conflict structure is the only variable.

Defaults and metrics. Table I lists the configuration every experiment uses unless the text states otherwise. The worker count is the one at which every engine peaks on this machine (§VI-B); the cap on group size rarely binds, since groups average 4.2 blocks on real Ethereum and 1.3 on the synthetic workload. Speedup is over single-threaded sequential `revm` on the same blocks, re-measured alongside each parallel run, and covers the execution phase only; §VI-E charges the preparatory stages. On real Ethereum the 15 rounds in which the Block-STM engine abandoned an optimistic window and re-executed it sequentially (an implementation fallback) are excluded from every statistic, leaving 135 rounds.

B. Execution Speedup

This subsection answers the first question, how much faster Omakase executes a round than the per-block engines and the concatenated round; Table II measures every engine of §VI-A in one run over the same rounds.

Comparing with per-block engines. Three of the engines in Table II execute one block at a time. On real Ethereum, Block-STM reaches 1.89× speedup. Graph-aware OCC reaches only 1.66×, although it has a conflict graph that Block-STM lacks. The graph is a prediction made by pre-executing the block, and a transaction that actually runs can touch keys it did not touch then, or skip keys it did. A missing edge costs a re-execution, and a spurious edge makes a transaction wait for a parent it does not need. Vegeta is the slowest at 1.33× because it runs one batch in parallel and waits for all of it to finish before starting the next. Thus, in Vegeta, a block with long dependency chains needs many batches, and at the end of each batch most workers sit idle. Omakase uses the same imperfect graphs as graph-aware OCC, but reaches 2.47×, 31% more than Block-STM and faster than it on 127 of the 135 rounds, and its slow blocks are faster too: the p99 per-block latency is 5.50 ms against 9.76 ms for Block-STM. The difference is that Omakase schedules multiple blocks together, so workers always have more independent transactions to pick from than one block contains.

On the synthetic workload the hints are exact. Block-STM reaches 2.02×, graph-aware OCC 2.08×, Vegeta 1.37× and Omakase 2.28×. With exact hints graph-aware OCC never re-executes a transaction. Block-STM, however, still aborts and retries on its own (0.73 re-executions per transaction). Omakase again adds the inter-block parallelism that no per-block engine provides.

TABLE III: Influence of the number of blocks per round.

Blocks per round B	20	50	100
Block-STM	1.83	1.86	1.89
Concatenated Block-STM	**2.31**	**2.32**	2.40
Omakase	2.19	**2.32**	**2.47**

Comparing with the concatenated round. The strongest baseline is also the simplest: treat the 100 blocks of a round as one big block and run Block-STM on it. This spreads every fixed per-block cost over the round and gives the optimistic engine the largest possible pool of transactions. It is Omakase with no limit on group size and no hot-resource test, and Omakase does not beat it everywhere.

The concatenated round wins in two situations. First, when the hints are exact. On the synthetic workload no graph-aware OCC engine ever re-executes a transaction, and graph-aware OCC run over the concatenated round reaches 2.99× speedup against 2.28× speedup for Omakase. On the artificial workload it is the best engine at every conflict level (§VI-C). Second, when the number of blocks per round is small. Over the real Ethereum workload it leads at 20 blocks per round, 2.31× against 2.19×, ties at 50 and loses at 100 (Table III). A 20-block window already can fully utilize the eight workers, so round size changes little for the concatenated window. For Omakase it changes how many balanced integrations the integrator can find, since it only integrates blocks that pass the hot-resource and balance tests, where we observed that the blocks per group grow from 2.9 blocks on average at 20 blocks per round to 4.2 at 100.

Omakase wins where one large optimistic window becomes fragile, which happens on the real Ethereum workload for two reasons. First, the hints are imperfect, so conflicts between transactions are discovered only at validation time. In the concatenated round, a transaction that reads a key written by an earlier block of the round can run before that write is done. Then it fails validation and re-executes, and each re-execution can invalidate the transactions that read its results. Based on our measurements, this happens 0.88 times per transaction, 61% more than for Block-STM running block by block, which does not validate across blocks. Omakase integrates only blocks that pass the hot-resource and balance tests into one group and executes groups one after another, so a conflict between two groups does not need validation. Inside a group, the integrated graph holds a transaction back until its RAW parents have been executed. It turns out to re-execute 0.31 times per transaction and runs 3% faster (2.47× against 2.40×) than the concatenated round with block-STM. Giving the concatenated round a conflict graph does not fix this. It cuts the re-executions by a third but has a lower speedup of 2.17×, because over a 15,000-transaction group the predicted edges make too many transactions wait. Second, Omakase performs more stably across rounds. Its p99 per-block latency is 5.50 ms against 8.45 ms for the concatenated round (p90 4.35 ms against 5.40 ms). The slow rounds are the ones with many conflicts between blocks. In one large window every such conflict adds re-executions that can spread through the whole window, so a highly conflicted round takes much longer than a typical one. Omakase's groups are small, and each block is integrated only if it shares no hot resource with the integrated graph, which limits the impact of a bad block to its own group.

TABLE IV: Per-round speedup with real-Ethereum workloads.

	speedup over sequential, per round		
Engine	slowest round	slowest tenth	slowest round/median
Block-STM	1.36	1.63	0.69
Concatenated Block-STM	1.51	1.93	0.61
Concatenated graph OCC	1.43	1.92	0.67
Graph-aware OCC	1.27	1.47	0.74
Vegeta	0.84	1.24	0.63
Omakase	**1.86**	**2.09**	**0.81**

TABLE V: The five rounds on which the concatenated round is slowest: speedup over sequential and re-executions per transaction.

	Block-STM		Concatenated		Omakase	
Round starting block	speedup	re-exec	speedup	re-exec	speedup	re-exec
18,582,826	1.36	0.54	1.51	0.90	**3.11**	**0.33**
18,583,826	1.37	0.72	1.59	1.22	**2.70**	**0.36**
18,586,026	1.54	0.55	1.69	1.15	**2.90**	**0.39**
18,586,226	1.61	0.64	1.73	1.06	**2.46**	**0.41**
18,582,026	1.51	0.67	1.78	1.18	**2.60**	**0.44**

What each design buys. When the hints are exact, the concatenated round has the higher average throughput. What Omakase buys instead is a per-round execution time that stays stable no matter how conflicted the round is, and that is what *congestion control* in our title means. An optimistic window with no size limit is fast on average but slower on a round with many conflicts between blocks.

Table IV looks at the 135 real-Ethereum rounds one by one. The concatenated round's slowest round runs at 1.51× speedup against 1.86× for Omakase, and the slowest tenth of its rounds run at 1.93× or less, against 2.09×. Its slow rounds are also further below its typical ones: its slowest round reaches 61% of its median round's speedup, against 81% for Omakase.

Table V lists the five rounds on which the concatenated round is slowest: it runs at 1.51–1.78× with 0.9–1.2 re-executions per transaction, Block-STM is no better, and Omakase runs at 2.5–3.1× with 0.3–0.4 re-executions. The reason is that Omakase fixes its groups before execution, and admits a block to a group only if it shares no hot resource with the group as integrated so far, so a round with many conflicts between blocks gets smaller groups instead of more re-executions. A validator must execute every round before the next, so the slow rounds determine its pace. Omakase gives up some average throughput for a stable per-round time.

C. When Does Integration Pay Off?

This subsection answers the second question: under which conflict pattern and parameter settings integration pays off. The artificial workload of §VI-A allows us to customize

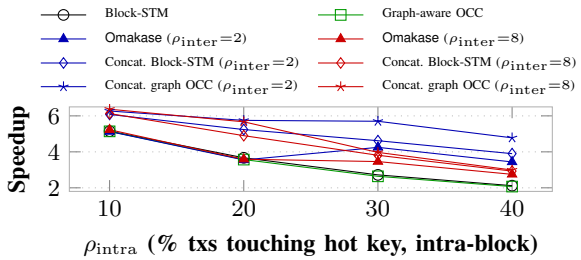


Fig. 6: Artificial workload, hot set H=5. Block-STM and graph-aware OCC depend only on ρ_{intra}; the three inter-block engines are shown for ρ_{inter}=2 (blue) and ρ_{inter}=8 (red).

the conflict pattern. We sweep $\rho_{\text{inter}} \in \{2, 4, 6, 8\}$ and $\rho_{\text{intra}} \in \{10, 20, 30, 40\}\%$ over 100 rounds of 50 blocks of 100 transactions, with a hot set of H=5 keys and the default Omakase parameters. Block-STM and graph-aware OCC execute each block on its own, so their speedup depends on ρ_{intra} alone.

Intra-block conflict (ρ_{intra}). Figure 6 shows that conflict inside a block is what hurts most. When 10% of a block's transactions write the hot key, the two per-block engines reach 5.2×; when 40% do, only 2.1×, since those transactions must run one after another. Up to 20% Omakase is no faster: each block alone keeps all eight workers busy, so the balance test sees no gain from integration. From 30% on a single block runs out of independent transactions, Omakase integrates blocks, and the groups win back most of the loss: 4.25× against 2.7× at 30%, and 3.44× against 2.1× at 40% (ρ_{inter}=2).

Inter-block conflict (ρ_{inter}). Conflict between blocks only matters once blocks are integrated, and then it eats into the gain: at ρ_{intra}=40%, Omakase falls from 3.44× when two blocks in a row share hot key h_1 to 2.75× when eight do: a round of 50 blocks runs as 22 groups at ρ_{inter}=2 and 32 at ρ_{inter}=8, which reduces the inter-block parallelism available within each group.

Why the concatenated round wins here. The concatenated round is the fastest engine in Figure 6. On the artificial workload the hints are exact, so concatenated graph OCC never re-executes a transaction, and one window of 5,000 transactions offers more parallelism than Omakase's 20∼30 integrate groups per round. However, we still do not consider it the better design: §VI-B showed that with the imprecise hints of real Ethereum, conflicting blocks congest a large window with re-executions (0.88 per transaction) and its slowest round falls to 1.51× against 1.86× for Omakase. Also, even in Figure 6, its advantage gets smaller as conflict grows, to 2.93× and 2.99× against 2.75× at $(\rho_{\text{inter}}, \rho_{\text{intra}})$=(8, 40%). As we can observe, Omakase is more resilient against the increasing inter-block conflict.

Hot-resource threshold (τ_{hot}). Figure 7 measures the influence of both thresholds in Omakase. For the hot-resource threshold, speedup rises from 2.25× at τ_{hot}=0.5 to 2.43× at τ_{hot}=1.0, and then stays flat up to τ_{hot}=5.0 (2.47×). A low threshold labels too many keys as hot, so the integrator refuses integrations that would have improved parallelism and leaves a 100-block round in 68 groups when τ_{hot}=0.5 instead of 24 when τ_{hot}=1.5. We then set τ_{hot}=1.5 by default.

TABLE VI: Abort accounting per transaction on real Ethereum workloads. *Cascade* = validation failure caused by a lower-indexed transaction's *re-execution*. *From another block*: validation failures whose invalidating writer is in a different block of the same group.

		validation failures		from another block	
Engine	re-exec	all	cascade	all	cascade
Block-STM	0.545	0.183	0.135	–	–
Concatenated Block-STM	0.877	0.208	0.193	–	–
Concatenated graph OCC	0.545	0.294	0.224	0.088	0.073
Graph-aware OCC	**0.268**	**0.123**	**0.080**	0	0
Omakase	0.309	0.151	0.101	0.082	0.056

Balance threshold (τ_{CV}). The integrator stops integrating once a group's simulated load is balanced, i.e., with a coefficient of variation below τ_{CV} (§V-D). A low threshold keeps integrating up to the cap, which we set to 16 groups per round at 0.1, for 2.49× speedup. A high threshold, however, declares groups balanced too early, with 97 groups per round at τ_{CV}=2.0, so speedup falls to 2.12×, close to per-block execution. We then set τ_{CV}=0.5 by default, which gives 2.47×.

Workers (k). Figure 7 (right) varies the worker thread count. Every engine is fastest with 8 workers on this machine, and all of them slow down when there are more than 8 workers. The two Block-STM-based engines fall faster than the graph-aware OCC engines. With more workers, more conflicting transactions execute optimistically and concurrently, but are then re-executed for validation failure. For example, the concatenated round re-executes transactions 1.60 times per transaction at k = 32 workers against 0.90 at k = 8. The three graph-aware OCC engines fall more gently because the graph keeps transactions from starting early. What remains is the cost of coordinating more workers, which still makes the engines slower.

D. Why Omakase Works: Fewer Aborts

This subsection answers the third question: why Omakase works, in terms of aborts. Table VI counts, per transaction, how often each engine re-executes a transaction and why, on real Ethereum workloads.

Two kinds of abort. A transaction is re-executed either because it failed validation or because, during execution, it read the placeholder of a transaction still being re-executed, or a nonce not yet in place, and gave up at once. Table VI counts every execution after the first as a re-execution and only the first kind as a validation failure. In optimistic execution a transaction fails validation when it read a value that an earlier transaction later changed. This happens in two ways. In an *ordinary abort* the earlier transaction simply had not

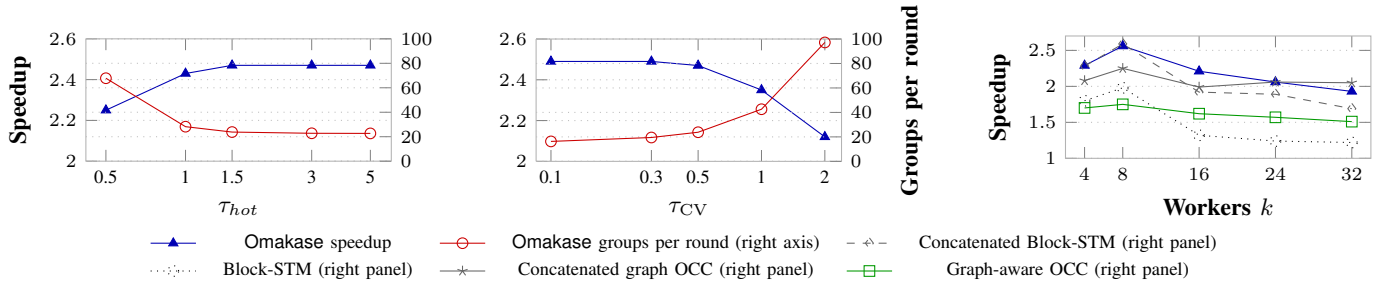


Fig. 7: Influences of the hot-resource threshold τ_{hot} (left), the balance threshold τ_{CV} (middle), and the worker count k (right).

run when the later transaction read. In a *cascade abort*, the earlier transaction had already run and was then re-executed. A re-execution creates a new version of every key it writes, and validation compares versions, not values, so a transaction that was placed correctly after the first run still fails even if the value did not change. Cascades are what the argument of §IV is about: one re-execution can invalidate the transactions that depend on it, and so on.

Cascades are the common case. As shown in Table VI, three quarters of Block-STM's validation failures are cascades, and for the concatenated round with block-STM the share is 93%, because putting a whole round in one optimistic window turns conflicts between blocks into a concurrent execution that can cascade.

The conflict graph removes most of them. A graph-aware engine does not start a transaction until its RAW parents have executed. Running block by block, graph-aware OCC has 41% fewer cascades and 51% fewer re-executions than Block-STM. Omakase puts several blocks in one group, which brings their conflicts between blocks back into play, yet it still has 25% fewer cascades and 43% fewer re-executions than Block-STM.

What is left. More than half of Omakase's remaining validation failures come from a writer in *another block* of the same group (0.082 of 0.151 per transaction, Table VI), and two thirds of those (0.056) are cascades: the reader was placed after the writer's first run, as it should be, but the writer was later re-executed and its new version invalidated the read. The graph orders first executions; it does not prevent a later re-execution from invalidating dependent readers. Group boundaries limit how far these cascades can propagate during concurrent execution.

E. Stage Costs and Validator Pipeline Throughput

Stage costs. This subsection answers the fourth question: what each stage of Omakase costs, and how much of that cost a validator's pipeline can hide by overlapping integration with execution. Table VII gives the cost of each stage per block. *Pre-execution* costs exactly one sequential run of the block. Every engine that uses hints pays it, but each proposer pays for only its own block, so with N proposers a validator's share is $1/N$ of it. Building the graph and reordering the transactions add another fifth of a sequential run. *Integration* is the one stage that only Omakase and concatenated round with graph-OCC have, and every validator pays it. With state in memory it is expensive, taking 2.91 ms per block on real Ethereum workloads. A validator that integrates and then executes gets 1.38× speedup only, below Block-STM's 1.89×. Adding $1/N$ of the proposer stages gives the "every stage on one node" row: with one block per proposer and N=100, a validator pays 6.73 ms per block in all, a speedup of 1.36× on real Ethereum and 1.85× on the synthetic workload.

TABLE VII: Cost of every stage per block (ms), as a fraction of sequential execution, and the resulting speedups.

	Real Ethereum		Synthetic	
Stage	ms/block	of seq.	ms/block	of seq.
Sequential execution (reference)	9.15	1.00	13.16	1.00
Pre-execute (proposer)	9.32	1.02	13.38	1.02
Build graph + intra-block reorder (proposer)	2.02	0.22	1.57	0.12
Integrate (validator)	2.91	0.32	1.18	0.09
Execute, Omakase (validator)	3.71	0.41	5.76	0.44
Execution only (Table II)	3.71	**2.47**×	5.76	**2.28**×
Validator: integrate + execute	6.62	**1.38**×	6.94	**1.89**×
Every stage on one node, N=100 proposers	6.73	**1.36**×	7.09	**1.85**×
Block-STM, no preparatory stage	4.83	1.89×	6.53	2.02×

TABLE VIII: Validator-side cost per block and speedup with the integration of round r+1 on one spare thread while round r executes.

	ms/block	speedup
Block-STM, block by block	5.03	1.91×
Omakase, execution only	4.09	2.34×
Omakase, integration only	3.31	–
Omakase, integrate then execute	7.40	1.30×
Omakase, integration overlapped with execution	4.29	**2.23**×

Hide cost by overlapping integration. Those rows charge integration and execution one after the other. A validator can instead integrate round r+1 on a spare core while its workers execute round r; Table VIII runs all 150 real rounds back to back both ways. Overlapping integration with execution raises the measured validator-side cost from 4.09 to 4.29 ms per block; the 0.20 ms increment corresponds to hiding approximately 94% of the 3.31 ms standalone integration cost. The resulting validator-side speedup is 2.23×, compared with 1.91× for Block-STM in the same run. These measurements cover validator-side preparation and execution; they do not measure transaction latency through proposal, dissemination,

and consensus.

VII. Conclusion

Multi-proposer consensus commits a batch of blocks every round, but today's parallel execution engines still run those blocks one at a time, so most of the parallelism in a round, and most of a validator's cores, are idle. This paper presented Omakase, which treats the batch as the unit of execution. Each proposer pre-executes only its own block and ships a conflict graph with it, so the cost of building conflict graphs is spread across all proposers. At the validator, a greedy integrator considers a block for addition to a group only when its hot-resource set is disjoint from that of the current integrated graph, and accepts the addition only if the balance test passes. A graph-aware OCC engine executes each integrated group, admitting a transaction only after its predicted RAW parents have executed. This reduces aborts caused by optimistic execution. Executing groups sequentially confines cross-block cascades to the current group, although re-executions can still trigger cascades within it.

On 15,000 Ethereum mainnet blocks, about 2.4 M transactions, Omakase executes a round 2.47× faster than sequential execution, against 1.89× for Block-STM and 1.66× for the same engine run block by block. The concatenated round, which treats the whole round as one optimistic window is about as fast on average (2.40×) but far less stable. Its slowest round runs at 1.51× against 1.86× for Omakase, and on the five rounds that it runs the slowest, Omakase is 1.4 to 2 times faster. The integration step that Omakase adds costs more than it saves if a validator waits for it, but overlapping integration with execution across rounds hides 94% of the measured integration cost in the validator pipeline.

In summary, Omakase exploits inter-block parallelism to achieve higher throughput than existing engines, and keeps stable performance when the inter-block conflicts are high. With both properties above, Omakase achieves congestion control in transaction execution for multi-proposer consensus.

Acknowledgments

Generative AI disclosure. In line with the IEEE policy on AI-generated content, we disclose that Anthropic's Claude (via Claude Code) was used under the authors' direction for the text, the figures and images, and the code of this work. No measurement, result or citation was generated by AI. The authors reviewed and verified every change and take full responsibility for the content.

References

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

[2] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, 2022, pp. 34–50.

[3] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 2705–2718.

[4] A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li, "Shoal: Improving DAG-BFT latency and robustness," *arXiv preprint arXiv:2306.03058*, 2023.

[5] B. Arun, Z. Li, F. Suri-Payer, S. Das, and A. Spiegelman, "Shoal++: High throughput dag bft can be fast!" *arXiv preprint arXiv:2405.20488*, 2024.

[6] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak, "Sailfish: Towards improving the latency of DAG-based BFT," Cryptology ePrint Archive, Paper 2024/472, 2024.

[7] S. Duan, H. Zhang, X. Sui, B. Huang, C. Mu, G. Di, and X. Wang, "Dashing and star: Byzantine fault tolerance with weak certificates," in *Proceedings of the nineteenth european conference on computer systems*, 2024, pp. 250–264.

[8] P. Jovanovic, L. Kokoris-Kogias, B. Kumara, A. Sonnino, P. Tennage, and I. Zablotchi, "Mahi-mahi: Low-latency asynchronous bft dag-based consensus," in *2025 IEEE 45th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2025, pp. 549–559.

[9] S. Xie, D. Kang, H. Lyu, J. Niu, and M. Sadoghi, "Fides: Scalable censorship-resistant dag consensus via trusted components," *arXiv preprint arXiv:2501.01062*, 2025.

[10] K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, and A. Sonnino, "Mysticeti: Low-latency DAG consensus with fast commit path," *CoRR*, vol. abs/2310.14821, 2023.

[11] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: resilient concurrent consensus for high-throughput secure transaction processing," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 1392–1403.

[12] D. Kang, S. Rahnama, J. Hellings, and M. Sadoghi, "SpotLess: Concurrent rotational consensus made practical through rapid view synchronization," in *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024*. IEEE, 2024.

[13] C. Stathakopoulou, T. David, and M. Vukolic, "Mir-BFT: High-throughput BFT for blockchains," 2019. [Online]. Available: http://arxiv.org/abs/1906.05552

[14] H. Lyu, S. Xie, J. Niu, I. Beschastnikh, Y. Zhang, M. Sadeghi, and C. Feng, "Orthrus: accelerating multi-bft consensus through concurrent partial ordering of transactions," in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 2025, pp. 2615–2627.

[15] H. Lyu, S. Xie, J. Niu, C. Feng, Y. Zhang, and I. Beschastnikh, "Ladon: High-performance multi-bft consensus via dynamic global ordering," in *Proceedings of the Twentieth European Conference on Computer Systems*, 2025, pp. 226–242.

[16] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks, "Autobahn: Seamless high speed bft," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, 2024, pp. 1–23.

[17] A. Ranchal-Pedrosa, D. Kang, N. Giridharan, D. Malkhi, M. Sadoghi, and B. Marsh, "Hermes: Low tail-latency via prefix consensus," *arXiv preprint arXiv:2607.25916*, 2026.

[18] A. Yakovenko, "Solana: A new architecture for a high performance blockchain v0. 8.13," *Whitepaper*, 2018.

[19] ——, "Sealevel — parallel processing thousands of smart contracts," https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, Feb. 2020, solana Labs Medium post, accessed April 2026.

[20] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino *et al.*, "Sui lutris: A blockchain combining broadcast and consensus," *arXiv preprint arXiv:2310.18042*, 2023.

[21] Sui Foundation, "All about parallelization," https://blog.sui.io/parallelization-explained/, 2024, sui Foundation blog post, accessed April 2026.

[22] "Aptos whitepaper: Safe, scalable, and upgradeable web3 infrastructure," 2022. [Online]. Available: https://arxiv.org/abs/2201.01107

[23] B. Marsh, S. Landers, and J. Jog, "Sei giga," *arXiv preprint arXiv:2505.14914*, 2025.

[24] Monad Labs, "Monad documentation: Parallel execution," https://docs.monad.xyz/monad-arch/execution/parallel-execution, 2024, accessed April 2026.

[25] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, February 25–March 1, 2023*. ACM, 2023, pp. 232–244.

[26] R. Neiheiser, A. Babaei, G. Alexopoulos, M. Kogias, and E. K. Kogias, "Chiron: Accelerating node synchronization without security trade-offs in distributed ledgers," 2026. [Online]. Available: https://arxiv.org/abs/2401.14278

[27] RISE Chain, "pevm: The ultimate parallel EVM engine," https://github.com/risechain/pevm, 2024, gitHub repository, accessed April 2026.

[28] D. Kang *et al.*, "Omakase: open-source implementation," https://github.com/DakaiKang/congestion_control, 2026, gitHub repository, accessed November 2026.

[29] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core oltp under high contention," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1583–1598. [Online]. Available: https://doi.org/10.1145/2882903.2882958

[30] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li, "Scaling multicore databases via constrained parallel execution," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1643–1658. [Online]. Available: https://doi.org/10.1145/2882903.2882934

[31] J. M. Faleiro, A. Thomson, and D. J. Abadi, "Lazy evaluation of transactions in database systems," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 15–26. [Online]. Available: https://doi.org/10.1145/2588555.2610529

[32] R. Johnson, I. Pandis, and A. Ailamaki, "Improving oltp scalability using speculative lock inheritance," *Proc. VLDB Endow.*, vol. 2, no. 1, p. 479–489, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687682

[33] H. Lyu, S. Xie, J. Niu, M. Sadoghi, Y. Zhang, C. Wang, I. Beschastnikh, and C. Feng, "Hydra: Breaking the global ordering barrier in multi-bft consensus," *arXiv preprint arXiv:2511.05843*, 2025.

[34] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *Proc. VLDB Endow.*, vol. 8, no. 11, p. 1190–1201, Jul. 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809981

[35] K. Ren, A. Thomson, and D. J. Abadi, "An evaluation of the advantages and disadvantages of deterministic database systems," *Proc. VLDB Endow.*, vol. 7, no. 10, p. 821–832, Jun. 2014. [Online]. Available: https://doi.org/10.14778/2732951.2732955

[36] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 70–80, Sep. 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920855

[37] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–12. [Online]. Available: https://doi.org/10.1145/2213836.2213838

[38] T. M. Qadah and M. Sadoghi, "QueCC: A queue-oriented, control-free concurrency architecture," in *Proceedings of the 19th International Middleware Conference, Middleware 2018, Rennes, France, December 10-14, 2018*, 2018, pp. 13–25.

[39] D. Qin, A. Demke Brown, and A. Goel, "Caracal: Contention management with deterministic concurrency control," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2021, pp. 180–194.

[40] The MystenLabs Team, "The sui smart contracts platform," https://docs.sui.io/paper/sui.pdf, 2023, accessed on 15.01.2024.

[41] Polygon Team, "Innovating the main chain: a polygon pos study in parallelization," https://polygon.technology/blog/innovating-the-main-chain-a-polygon-pos-study-in-parallelization, 2022, accessed on 05.12.2022.

[42] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 303–312. [Online]. Available: https://doi.org/10.1145/3087801.3087835

[43] H. Zhang, J. Li, H. Zhao, T. Zhou, N. Sheng, and H. Pan, "Blockpilot: A proposer-validator parallel execution framework for blockchain," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 193–202. [Online]. Available: https://doi.org/10.1145/3605573.3605621

[44] J. Silva, E. Alchieri, F. Dotti, and F. Pedone, "Parallel execution of transactions based on dynamic and self-verifiable conflict analysis," in *Proceedings of the 12th Latin-American Symposium on Dependable and Secure Computing*, ser. LADC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 110–119. [Online]. Available: https://doi.org/10.1145/3615366.3615425

[45] P. S. Anjana, M. Amini, R. Kapoor, R. Parmar, R. Ramesh, S. Ravi, and J. Tobkin, "Efficient Parallel Execution of Blockchain Transactions Leveraging Conflict Specifications," in *7th Conference on Advances in Financial Technologies (AFT 2025)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), Z. Avarikioti and N. Christin, Eds., vol. 354. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, pp. 29:1–29:26. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.AFT.2025.29

[46] T. Xu, Y. Zhong, Y. Zhang, R. Xiong, J. Zhang, G. Xue, and S. Liu, "Vegeta: Enabling parallel smart contract execution in leaderless blockchains," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. Philadelphia, PA: USENIX Association, Apr. 2025, pp. 795–811. [Online]. Available: https://www.usenix.org/conference/nsdi25/presentation/xu-tianjing

[47] J. Chen, A. Sonnino, L. Kokoris-Kogias, and M. Sadoghi, "Thunderbolt: Concurrent smart contract execution with non-blocking reconfiguration for sharded dags," 07 2025.

[48] R. Neiheiser and E. Kokoris-Kogias, "Anthemius: Efficient and modular block assembly for concurrent execution," in *Financial Cryptography and Data Security*, C. Garman and P. Moreno-Sanchez, Eds. Cham: Springer Nature Switzerland, 2026, pp. 307–323.

[49] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.

[50] V. Buterin, "Ethereum white paper: A next-generation smart contract and decentralized application platform," https://ethereum.org/en/whitepaper/, 2013.

[51] X. Sui, S. Duan, and H. Zhang, "Marlin: Two-phase BFT with linearity," in *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2022, pp. 54–66.

[52] Q. Kniep, L. Kokoris-Kogias, A. Sonnino, I. Zablotchi, and N. Zhang, "Pilotfish: Distributed execution for scalable blockchains," in *International Conference on Financial Cryptography and Data Security*. Springer, 2025, pp. 287–306.

[53] Z. Liu, A. Sonnino, I. Zablotchi, E. K. Kogias, and M. Kogias, "Remora: Scale-out deterministic execution for smart contracts," *arXiv preprint arXiv:2607.02817*, 2026.