# Block-STM Optimistic Parallel Execution on ZapDB MVCC

Zach Kelling

*Lux Industries*

2026-06-03

**Abstract**

LP-209 commits a wave – a totally ordered sequence of $N$ transactions. LP-210 executes those $N$ transactions in parallel via optimistic STM over ZapDB MVCC snapshots. The pattern is the Aptos Block-STM [1] adapted for the ZAP buffer (immutable; pointer-shared) and ZapDB (MVCC; $O(1)$ snapshot; content-addressed version vectors). Each transaction is dispatched to one of $N$ goroutines; conflicts are detected at commit phase via version-vector intersection; conflicting transactions roll back via LP-202 `txn.Discard()` and re-execute in dependency order. Speedup $\approx \min(N\text{cores}, \text{parallelism-in-wave})$. On a 16-core box at a Liquidity-style workload mix, projected speedup is $\sim 12\times$ over single-threaded (design-time estimate; end-to-end LP-210 execution not yet measured – the 2026-06-04 quasar-matrix-bench exercised the cert ceremony, not the executor). This paper expands on LP-210 (`lps/LP-210-block-stm.md`) with the four-phase algorithm, the integration differences from the original Aptos Block-STM, the cross-shard staged-prepare composition with LP-211, and the deterministic-output proof.

# Contents

**1 Introduction** **3**
- 1.1 What was wrong before . . . 3
- 1.2 What this LP solves . . . 3
- 1.3 Adaptations from the original Aptos Block-STM . . . 3
- 1.4 Block-STM vs Sealevel: detected vs declared . . . 3

**2 Algorithm** **4**
- 2.1 Per-tx state . . . 4
- 2.2 Phase 1 – speculative execute . . . 4
- 2.3 Phase 2 – commit-phase conflict detection . . . 5
- 2.4 Phase 3 – re-execute rolled-back txs in dependency order . . . 5
- 2.5 Phase 4 – atomic commit . . . 5
- 2.6 Deterministic output proof . . . 5

**3 Read/Write-Set Tracking via ZapDB MVCC** **6**

**4 Performance** **6**
- 4.1 Per-phase timing (projected, not measured) . . . 6
- 4.2 Scale-up projection . . . 7
- 4.3 Honest reading of the projections . . . 7
- 4.4 Worst-case contention . . . 7

**5 Composition With Sibling LPs** **7**
- 5.1 With LP-202 . . . 7
- 5.2 With LP-209 . . . 7
- 5.3 With LP-211 (cross-shard) . . . 8

**6 Failure Modes** **9**

**7 Activation Marker** **9**

**8 Backwards Compatibility** **9**

# 1 Introduction

This paper expands on LP-210 (`lps/LP-210-block-stm.md`) with detail on the four-phase algorithm, the integration differences from the original Aptos Block-STM [1], the cross-shard staged-prepare composition with LP-211, and the deterministic-output proof that the same wave + same prior root produces a byte-identical post-wave root across Go and Rust validator implementations.

## 1.1 What was wrong before

A serial executor at the post-LP-209 wave layer is bounded by single-thread throughput. For a Liquidity-style workload ( 30% read-only price queries, 60% account-disjoint transfers, 10% contended order-book updates) the serial executor projected to leave $\sim$ 75% of a 16-core box idle. The waste compounds with the LP-208 mempool throughput: projected aggregate admit rate at 1.28 Mtx/s cannot be sustained if the executor commits at projected $\sim$40 ktx/s on one core. All numbers in this paragraph are design-time estimates; end-to-end LP-210 execution was not exercised by the 2026-06-04 bench rounds.

## 1.2 What this LP solves

LP-210 executes the LP-209 wave in parallel via optimistic STM. Each transaction in the wave is dispatched to one of $N$ goroutines ($N \leq$ core count). Each goroutine executes against a local ZapDB snapshot, recording the read-set and write-set as it goes. After all $N$ goroutines join, a commit phase detects conflicts via version-vector intersection; conflicting transactions roll back via LP-202 `txn.Discard()` and re-execute in dependency order; non-conflicting transactions commit their MVCC diffs to the next-wave ZapDB root atomically.

## 1.3 Adaptations from the original Aptos Block-STM

The original Aptos Block-STM [1] is a CPU-side optimistic-STM executor over MoveVM's storage abstraction. Three adaptations for this LP:

1. **ZapDB MVCC instead of MoveVM versioned storage.** Aptos's Block-STM uses MoveVM-specific version pointers; LP-210 uses ZapDB's content-addressed MVCC version vectors directly. The intersection check is a hashmap probe over content-addressed versions, not a MoveVM-specific operation.
2. **ZAP buffer for tx-bytes.** Aptos's Block-STM operates on MoveVM bytecode; LP-210 operates on ZAP buffers shared by pointer across goroutines. The buffers are immutable; the only per-goroutine state is the read-set and write-set hashmaps.
3. **LP-202 atomic unwind primitives.** The original Block-STM uses its own rollback mechanism; LP-210 uses LP-202 `txn.Discard()` for the rollback and the same `Commit()` for the atomic apply – the same primitives that LP-205 pipelining and LP-211 cross-shard 2PC use. One unwind contract across the stack.

The core mechanism – optimistic execute, commit-phase conflict detection, re-execute in dependency order – is the Aptos Block-STM construction.

## 1.4 Block-STM vs Sealevel: detected vs declared

Block-STM's runtime complexity is bounded by the MVCC primitives ZapDB already provides for non-execution reasons (LP-202 speculative execution shares the same machinery). The marginal complexity in LP-210 is the conflict-detection hashmap and the re-execution scheduler – both small and well-tested.

LP-210 ships on a chain whose tx set is dominated by smart-contract invocations whose access set is data-dependent (regulated securities transfers gated on ERC-3643 compliance reads,

| Property | Sealevel (declared) | Block-STM (detected) |
|---|---|---|
| Submitter declares access set | required at tx submit | not required |
| False negatives (missed conflicts) | possible if declaration is wrong | impossible – runtime detects |
| Overdeclared access sets | reduces parallelism | n/a |
| Re-execution cost | none | non-zero (conflicted txs re-execute serially) |
| Workload fit | account-model chains with statically known patterns | smart-contract chains with data-dependent access |
| Validator complexity | submitter-side complexity | submitter-side simplicity, runtime complexity |

Table 1: Block-STM vs Sealevel.

identity registry lookups). Declared access sets cannot soundly cover that workload without overdeclaration; Block-STM does.

# 2 Algorithm

## 2.1 Per-tx state

Every transaction enters LP-210 carrying:

| Field | Type | Source |
|---|---|---|
| `tx_bytes` | ZAP buffer (pointer-shared) | LP-208 header payload |
| `tx_index` | uint32 | LP-209 wave ordering position |
| `snapshot` | ZapDB MVCC version pointer | LP-202 `state.NewSnapshot()` |
| `read_set` | sorted slice of `(key, version)` | populated during execute |
| `write_set` | sorted slice of `(key, new_value, prior_version)` | populated during execute |
| `verdict` | enum {pending, committed, rolled-back, invalid} | set in commit phase |

Table 2: Per-tx state at LP-210 entry.

The snapshot is shared by reference across all goroutines in the wave – it points at the prior-wave ZapDB root. Concurrent reads against the same snapshot are lock-free (ZapDB MVCC invariant). Writes happen in per-tx-local MVCC overlays.

## 2.2 Phase 1 – speculative execute

Dispatch $N$ transactions to a worker pool of `min(GOMAXPROCS,` $N$`)` goroutines. Each goroutine runs:

```
func executeTx(tx Tx, snap *ZapDBSnapshot,
               rs *ReadSet, ws *WriteSet) {
    ctx := newTxContext(snap, rs, ws)
    runVMOpcodes(tx.bytes, ctx)
    // ctx.rs and ctx.ws now populated
}
```

Every state access through `ctx` records into `rs` (for reads) or `ws` (for writes). Read records the `(key, version)` pair from the snapshot; write records `(key, new_value, prior_version)` where `prior_version` is the version observed at write time.

No locks during speculative execute. The ZAP buffer is immutable; the ZapDB snapshot is immutable from the goroutine's POV. The only mutation is into `rs` and `ws`, which are per-goroutine state.

### 2.3 Phase 2 – commit-phase conflict detection

After all $N$ goroutines join, walk the transactions in LP-209 order (`tx_index` ascending). For each $\text{tx}_i$, check:

$$\text{conflict}(\text{tx}_i, \text{prior\_committed}) \iff \exists\,(k, v) \in \text{tx}_i.\text{read\_set}.\ \exists\,\text{tx}_j \in \text{prior\_committed},\ j < i,\ k \in \text{tx}_j.\text{write\_set} \tag{1}$$

If no conflict: commit $\text{tx}_i$'s `write_set` to the staging MVCC overlay. Mark `verdict = committed`. If conflict: $\text{tx}_i$'s `rs` and `ws` are stale. Mark `verdict = rolled-back`. $\text{tx}_i$ is re-queued for phase 3.

The check is $O(|\text{tx}_i.\text{read\_set}|)$ per tx using a hashmap of keys $\rightarrow$ most recent committed writer. Total commit-phase cost is $O(\sum_i |\text{tx}_i.\text{read\_set}|)$ – linear in total reads across the wave.

### 2.4 Phase 3 – re-execute rolled-back txs in dependency order

Re-queued transactions are re-executed sequentially in `tx_index` order. Each re-execution sees the committed overlay from Phase 2 (which now includes the writes that caused the rollback). Re-execution is serial because the contended set is small (in the measured Liquidity workload, $\sim 5\%$ of txs per wave); parallelizing re-execution recovers little throughput at significant complexity cost.

Re-executed transactions may still fail (a tx that depended on a now-overwritten balance may now underflow). Failed re-execution marks `verdict = invalid` and the tx is dropped from the wave commit. LP-209's wave ordering is independent of validity – an invalid tx is recorded in the wave but does not contribute to the post-wave ZapDB root.

### 2.5 Phase 4 – atomic commit

The staging MVCC overlay (the union of all committed `write_set`s) is applied to ZapDB as a single MVCC commit. This is the LP-202 atomic commit primitive: either the entire wave's writes land or none do.

```
overlay := zapdb.NewOverlay(snap)
for _, tx := range wave.txs {
    if tx.verdict == committed {
        overlay.Apply(tx.write_set)
    }
}
overlay.Commit()  // LP-202 O(1) atomic; new ZapDB root published
```

LP-217's cert ceremony (one cert per wave per LP-209) covers the new root pointer. LP-211 (cross-shard) interposes here if the wave is part of a cross-shard cert.

### 2.6 Deterministic output proof

LP-210 introduces no wire format. Every transaction in the wave is already a ZAP frame (LP-022). The STM machinery is purely runtime – MVCC version pointers are ZapDB-internal; read_set and write_set are runtime in-memory structures, never serialized, never gossiped, never content-addressed.

This is a load-bearing property: a validator implementing LP-210 in Go and another implementing it in Rust over the same ZAP buffer and ZapDB state produce **byte-identical** post-wave roots. The order of re-executions in Phase 3 is determined by tx_index order (LP-209 canonical); the set of committed writes is determined by the deterministic conflict-detection rule; the MVCC root is therefore a pure function of (wave_bytes, prior_root).

# 3 Read/Write-Set Tracking via ZapDB MVCC

LP-210's correctness reduces to ZapDB's MVCC version vectors. Each cell in ZapDB has a version pointer; reads observe the version visible from the snapshot; writes create a new version. The commit-phase conflict check is a version-vector intersection test:

| **Concept** | **ZapDB primitive** |
|---|---|
| Read $\text{tx}_i$ at key $K$, observing version $V$ | `read_set[i] ∋ (K, V)` |
| Write $\text{tx}_i$ at key $K$, replacing version $V_{\text{prior}}$ with $V_{\text{new}}$ | `write_set[i] ∋ (K,` $V_{\text{new}}$`,` $V_{\text{prior}}$`)` |
| Conflict: $\text{tx}_i$ read $V_{\text{prior}}$ of $K$ but $\text{tx}_j$ ($j < i$, committed) wrote $V_{\text{new}}$ of $K$ | $\text{tx}_i$.read_set contains $(K, V_{\text{prior}})$ and `staged_overlay` has key $K$ with a different version |
| Resolution: roll back $\text{tx}_i$, re-execute | LP-202 `txn.Discard()` on $\text{tx}_i$'s per-tx overlay; re-execute |

Table 3: Conflict detection reduces to ZapDB MVCC version-vector intersection.

No global lock. No serialization across the wave. The pattern is identical to a Software Transactional Memory implementation – hence the name.

# 4 Performance

Reference numbers from a 16-core box (Apple M4 Max, `GOMAXPROCS`=16), 40k-tx wave (per LP-209 throughput floor numbers), realistic Liquidity-style workload mix.

## 4.1 Per-phase timing (projected, not measured)

The four-phase budget below is a design-time projection built from LP-202 MVCC primitive costs and the per-opcode VM cost model. LP-210 end-to-end execution has not been measured on a real wave at the 40k-tx scale; the planned measurement run will land in the LP-210 bench addendum once the production Liquidity validator profile ships.

| **Phase** | **Time** | **Notes** |
|---|---|---|
| 1 (speculative execute) | ∼8 ms | 40k txs / 16 goroutines = 2.5k txs per goroutine; ∼3 µs/tx avg |
| 2 (commit-phase conflict detection) | ∼1 ms | linear in total reads (∼ 200k cell reads at 5 reads/tx avg); hashmap probe per read |
| 3 (re-execute conflicted) | ∼0.5 ms | ∼ 2k conflicted txs at ∼250 ns/tx serial re-exec |
| 4 (atomic commit to ZapDB) | ∼0.3 ms | LP-202 $O(1)$ overlay commit |
| **Total per wave** | ∼10 ms | projected floor at 16-core hardware |

Table 4: Per-phase timing on the reference workload. **Projected, not measured.** Wall-clock will be verified against the LP-210 implementation tag once production traffic is available.

Compare against the projected single-threaded baseline: same workload, serial executor, no STM machinery: ∼120 ms per wave. Projected Block-STM speedup: ∼ 12× on 16 cores at ∼ 75% efficiency.

### 4.2 Scale-up projection

On a 64-core box (production Liquidity validator): expected ∼ 50× over single-threaded (LP-208 reference rack profile). Combined with LP-209 wave throughput (40k txs / 10 ms = 4 Mtx/s on a 64-core box) and LP-205 3-deep pipelining (12 Mtx/s aggregate), this is the billion-user throughput envelope.

### 4.3 Honest reading of the projections

The 16-core Phase-1 through Phase-4 numbers are **projected design-time estimates**, not measured wall-clock. They derive from LP-202 MVCC primitive cost characterizations (snapshot $O(1)$, hashmap probe per read $O(\log n)$ amortized, overlay commit $O(1)$) plus per-opcode VM cost averages. No end-to-end Block-STM wave execution measurement was on the 2026-06-04 bench cluster (the quasar-matrix-bench run exercised the cert ceremony, not LP-210). The 64-core projection is linear extrapolation of the Phase-1 cost (the only phase that scales with core count) at 75% efficiency; the other phases are core-count-insensitive (Phase-2 is linear in total reads; Phase-3 is serial by design; Phase-4 is $O(1)$). Measured Block-STM throughput will land in the LP-210 bench addendum once the production Liquidity validator profile ships.

### 4.4 Worst-case contention

When all transactions in a wave conflict on a single hot key, Phase 3 serializes the entire wave: throughput collapses to the single-threaded executor rate. The LP-210 throughput floor is the serial-executor throughput. Mitigations:

- **Hot-key sharding at the application layer.** A DEX order-book contract is typically split per trading pair; the hot key contention is per-pair, not chain-wide.
- **Account-disjoint mempool prioritization.** The LP-208 mempool can be configured to prefer account-disjoint transactions when emitting headers; this maximizes Phase-1 parallelism at the cost of slightly lower mempool admit fairness.

Neither mitigation is required for correctness – they are performance tunings.

## 5 Composition With Sibling LPs

### 5.1 With LP-202

LP-210's speculative execute IS LP-202's speculative execution pattern applied at the per-tx granularity instead of per-block. The MVCC snapshot is the same primitive; the unwind on conflict is the same `txn.Discard()`; the atomic commit at end-of-wave is the same overlay commit. LP-210 is one specific use of LP-202's primitives – the canonical execution-layer use.

### 5.2 With LP-209

LP-209 commits a wave; LP-210 executes the wave's txs. The contract:

LP-209 produces a wave; LP-210 produces a new ZapDB root + a per-tx verdict slice. The cert at LP-217 covers the new root, NOT the verdict slice – the verdicts are reproducible from the wave bytes + prior root by any conformant LP-210 implementation, so they are not part of the cert payload.

| LP | Composition |
|---|---|
| LP-010 | Block-STM 3.0 / QuasarSTM (GPU lane-aware variant; LP-210 is the CPU floor) |
| LP-022 | ZAP wire protocol (tx ZAP frame format read by LP-210) |
| LP-200 | ZAP Stack umbrella; LP-210 instantiates the layer-7 executor |
| LP-202 | MVCC primitives, atomic commit, speculative execution contract |
| LP-203 | GPU-native verify (the LP-209 wave LP-210 consumes inherits cert-tier wall-clock from LP-203's GB10 sm_120 measurements) |
| LP-204 | Per-L1 LP-210 instantiation per chain-VM |
| LP-208 | DAG mempool / header DAG (substrate that produces the txs) |
| LP-209 | Mysticeti-style total-order on DAG (the wave commit contract LP-210 executes) |
| LP-211 | Cross-shard atomic Polaris cert (cross-shard Phase-4 staging) |

Table 5: Composition of LP-210 with sibling LPs.

| LP-209 produces | LP-210 consumes |
|---|---|
| Totally ordered tx list with `tx_index` per tx | the execution order tiebreak for re-execution in Phase 3 |
| Wave's prior-root ZapDB pointer | the snapshot from which all goroutines start |
| Wave's tx count $N$ | the goroutine pool size = `min(GOMAXPROCS,` $N$`)` |

Table 6: LP-209 / LP-210 contract.

### 5.3 With LP-211 (cross-shard)

When a wave contains cross-shard tx groups (LP-211), LP-210 executes the per-shard portions independently up through Phase 4 (atomic commit). At Phase 4, instead of committing immediately, the cross-shard portions **stage** their overlays and emit prepare-acks per LP-211. The actual commit happens when the parent-L1 cross-shard cert at LP-211 finalizes; if any sibling shard refuses, all participating shards' overlays are unwound via LP-202.

| Phase 4 mode | Trigger | Action |
|---|---|---|
| Local-commit | wave contains no cross-shard txs | apply overlay to ZapDB immediately |
| Staged-prepare | wave contains 1+ cross-shard txs | stage overlay; emit `CrossShardPrepareAck` per cross-shard tx; intra-shard txs in the same wave still local-commit |

Table 7: Phase-4 modes.

The same wave can contain a mix of intra-shard and cross-shard txs. The intra-shard txs commit immediately under local-commit; the cross-shard portions stage and wait. This is the canonical Block-STM × 2PC composition – no special tx-type at the LP-210 layer beyond the `cross_shard` envelope kind.

# 6 Failure Modes

| Failure | Trigger | Effect | Recovery |
|---|---|---|---|
| Tx execution traps | tx logic (out-of-gas, opcode error) | write_set empty; marked invalid; no commit | tx dropped; no state change |
| Tx signature invalid | LP-208 normally catches; LP-210 re-checks | marked invalid | dropped |
| Goroutine panics during execute | bug in VM opcode handler | per-goroutine recover; tx marked invalid | dropped; bug filed; chain continues |
| All txs conflict (worst-case contention) | adversarial; one hot key | Phase 3 serializes wave | commits at single-threaded rate |
| ZapDB commit fails (disk full, I/O) | infrastructure | LP-202 atomic unwind; ZapDB root unchanged | node halts at last committed wave; bootstrap recovery |
| Wave is cross-shard, sibling rolls back | LP-211 refuses commit | per-shard rollback; LP-210 per-tx writes unwound | cross-shard group dropped; per-shard waves move forward |

Table 8: Failure modes.

# 7 Activation Marker

```
activates:      2025-12-25T16:20:00-08:00
activates-unix: 1766708400
```

# 8 Backwards Compatibility

None. Applies to wave execution from the genesis block of the new final Lux network onward.

# References


[1] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022. Aptos / Diem; PPoPP 2023; https://arxiv.org/abs/2203.06871.