LP-010: QuasarSTM

Block-STM 3.0 — A GPU-Native Lane-Aware Ordered MVCC Fabric for Quasar-Certified Rounds

Zach Kelling

Lux Industries

Spec freeze: December 15, 2025 | Activation: December 25, 2025

Abstract

We present *QuasarSTM*, the third generation of Block-STM, designed as the ordered multi-version concurrency control (MVCC) fabric beneath Lux's Nova (linear) and Nebula (DAG) execution modes. Block-STM 1.0 (Aptos 2022) introduced ordered speculative execution on CPU. Block-STM 2.0 was the GPU wave-drain port, treating the GPU as a faster CPU. QuasarSTM 3.0 is a re-architecture: STM is no longer a retroactive conflict detector; it is a scheduler-aware fabric. Its primary abstraction is the *lane*—a deterministic hash of (domain, contract, account, asset, market, nonce-lane, storage-prefix)—which is the smallest independently schedulable state domain. Lanes drive Prism refraction (so STM rarely sees the same hot conflict twice), three-tier validation (lane-clock, key-MVCC, semantic commutativity), deterministic contention management, and multi-GPU sharding.

We give formal definitions of lanes, ordered MVCC visibility for both Nova and Nebula, three-tier validation, TicToc-style per-key timestamps that reduce false aborts without reordering consensus, checkpoint-based incremental repair, and the commit horizon. We prove four invariants—determinism, ordered serializability, repair monotonicity, and horizon safety—and report production-hardware measurements: a 1024-transaction end-to-end stress completes in ∼150 ms on Apple M1 Max in 8 wave ticks, and the canonical hot-key repair workload (16 same-key contending transactions) yields exactly 120 conflicts, 120 repairs, and 16 commits—textbook Block-STM running entirely on GPU. The headline target is $repair\_amplification < 1.01$ on normal regulated-DEX workloads, with controlled degradation rather than repair explosion under heavy contention.

Contents

1 Introduction

Block-STM [12] introduced ordered speculative execution to public blockchains: transactions in a block execute in parallel under their canonical position; conflicts are detected retroactively against MVCC version chains; offending transactions re-execute with bumped incarnation numbers until the read set stabilises. The contribution was conceptual. A known canonical order is not a serialization bottleneck; it is a validation primitive.

The Lux execution stack has shipped three generations of this idea.

1. **Block-STM 1.0** — a CPU port of the Aptos design (`evmgpu`'s Go reference). Multi-threaded ordered speculation, per-key version chains, retroactive validation. Adequate for a few thousand transactions per block, throughput-bound by L3 traffic and conflict re-execution.

2. **Block-STM 2.0** — the first GPU port. The GPU was treated as a faster CPU: the same monolithic execute–validate–repair loop, just with thousands of fibers in place of OS threads. A single MVCC arena. The wave-drain executor. Repair-rate telemetry. Production at `cevm` v0.30.

3. **Block-STM 3.0 / QuasarSTM** — a re-architecture, not a port. STM stops being a retroactive conflict detector and becomes a *scheduler-aware fabric*. Lanes are first-class. Validation is three-tier (lane-clock, key-MVCC, semantic). Repair is incremental against fiber checkpoints. Commit happens by prefix/cut, not transaction-by-transaction. Contention is managed deterministically, with explicit hard limits against SIMT livelock.

Headline change. Block-STM 1.0 and 2.0 *discover* contention. They run a workload, record where conflicts happened, and try again. QuasarSTM 3.0 *predicts* contention through lane hints, *prevents* the predictable cases through Prism refraction, and uses MVCC only where lane prediction is insufficient. When QuasarSTM is constantly repairing, the design has failed; the production target is $\mathit{repair\_amplification} = \mathit{repair\_count} / \mathit{committed\_count} < 1.01$ on normal regulated-DEX workloads.

Why now. Three forcing functions made the rewrite necessary. First, a regulated DEX workload (order-book appends, fee accumulation, balance deltas, audit appends) has dramatically more lane-local structure than naive ERC-20 traffic. Lane partitioning extracts that structure; key-only MVCC squanders it. Second, the Quasar consensus engine [14] requires *ordered, deterministic, GPU-resident* execution to feed QuasarCert root construction without leaving the device. The Quasar round result is a commitment over `block_hash`, `state_root`, `receipts_root`, `execution_root`, `mode_root`; a CPU-side STM cannot supply these on the wave-tick budget. Third, GPU-STM literature has long flagged SIMT livelock and unbounded retry as the dominant failure modes [10, 13]. A correctness-by-construction repair manager is mandatory, not optional.

The lane abstraction. The single most important change in 3.0 is that the unit of independence is no longer the storage key. It is the *lane*:

$$\mathit{lane\_id} = H(\mathit{domain}, \mathit{contract}, \mathit{account}, \mathit{asset}, \mathit{market}, \mathit{nonce\_lane}, \mathit{storage\_prefix})$$

Lanes are the unit of frontier partitioning, fast validation, hot-state detection, semantic reduction, repair priority, and multi-GPU sharding. Without lanes, Block-STM discovers contention too late. With lanes, Block-STM is a safety net beneath a primary scheduler.

1.1 Contributions

1. A formal model of ordered MVCC for both linear (Nova) and DAG (Nebula) consensus modes, parameterised over a mode-specific ordering oracle (§5).
2. The lane abstraction (§3) and lane clocks, together with a TL2-derived fast validation tier (§4) that admits commits without touching per-key version chains when lane state is undisturbed.
3. Three-tier validation: lane-clock, key-MVCC, and semantic commutativity (§4). Tier 3 is the difference between a fast blockchain and a fast regulated DEX.
4. TicToc-style per-key read/write timestamps adapted to a known canonical order (§6). Hard rule: timestamps reduce false aborts but cannot reorder consensus.
5. Prism refraction (§7) as the primary contention *prevention* mechanism, partitioning ready frontiers by lane conflict matrix before STM ever sees them.
6. Incremental repair against fiber checkpoints (§8), cutting repair cost by 5–30× on router and DEX workloads relative to full re-execution.
7. A deterministic contention manager (§9) with six fixed policies and explicit hard limits, eliminating the nondeterministic-victim and unbounded-retry pathologies catalogued in the GPU-STM literature.
8. Commit horizons (§10) that finalise contiguous Nova prefixes or valid Nebula causal cuts in batches, slashing per-transaction certificate overhead.
9. Four formal invariants (§12): determinism, ordered serializability, repair monotonicity, and horizon safety.
10. Production measurements (§13): 1024-tx end-to-end stress in ∼150 ms on M1 Max; hot-key contention test of 16 same-key transactions producing exactly 120 conflicts, 120 repairs, and 16 commits, matching textbook Block-STM behaviour entirely on GPU.

Scope. This paper specifies QuasarSTM 3.0 frozen as of December 15, 2025 and activated on the Quasar 3.0 mainnet on December 25, 2025. Work in progress on the adaptive evolution path (3.1 ConflictSpec / NEMO LaneClass / Aria, 3.2 RapidLane / ForeSight / Multiverse, 4.0 CSMV / Motor / vMVCC) is referenced in §14 but is explicitly out of scope for the 3.0 specification.

2 Background

QuasarSTM borrows from four lines of work—ordered serializability, Block-STM, TL2, and TicToc—and rejects the failure modes documented in the GPU-STM literature.

2.1 Ordered Serializability under a Known Canonical Order

Definition 2.1 (Canonical order)**.** *A consensus mode supplies a total order $\preceq_{\text{can}}$ over the transactions in a block (Nova) or a strict partial order with a deterministic linear extension over the ready cut (Nebula). The canonical order is fixed before execution and is independent of execution outcomes.*

Definition 2.2 (Ordered serializability)**.** *A history H over committed transactions $T_1, \ldots, T_k$ is* ordered-serializable *with respect to $\preceq_{\text{can}}$ iff there exists a serial schedule S equivalent to H such that S respects $\preceq_{\text{can}}$.*

Block-STM's central observation is that when a canonical order is already fixed by consensus, validation does not need to discover one. It only needs to verify that each committed read observed the version its canonical predecessor produced. This collapses the validation problem from $\Theta(k!)$ schedules to a per-read version equality.

2.2 Block-STM (Aptos)

Block-STM [12] maintains a per-key version chain $V(k) = \langle v_0, v_1, \ldots \rangle$ keyed by canonical transaction index. Transactions execute speculatively, recording reads and writes. Validation walks the read set: each read at canonical position i must observe the version written by the largest $j < i$ that wrote k, evaluated against the latest valid incarnation. Mismatches trigger re-execution at incarnation $\iota + 1$. Convergence is by induction on i: once all T_j for $j < i$ are stable, re-executing T_i at most once produces a valid commit.

QuasarSTM keeps this skeleton. It modifies what *counts as a read mismatch* (lane clocks short-circuit it; semantic reducers redefine it) and what *counts as a repair* (incremental rollback against fiber checkpoints replaces full re-execution).

2.3 TL2: Transactional Locking II

TL2 [11] introduced commit-time locking with a global version clock. Each transaction snapshots the clock at start and validates by checking that no observed object's version exceeds the snapshot. The advantage is invisible reads on the fast path; the disadvantage is that a single global clock becomes a contention point under high write throughput.

QuasarSTM *shards* the TL2 idea. There is no global clock. Each lane has its own *lane clock* (§4). A transaction that touches lanes $L_1, \ldots, L_m$ snapshots $\langle c_1, \ldots, c_m \rangle$ and validates by checking that no c_i has advanced. This is the Tier 1 fast path: under low contention, transactions commit without touching the per-key MVCC chains at all.

2.4 TicToc: Per-Data-Item Timestamps

TicToc [19] eliminates the centralised timestamp counter by storing a $(read\_ts, write\_ts)$ pair per data item and computing each transaction's commit timestamp *lazily* as the smallest value compatible with its read and write footprint. The result is throughput close to ideal optimistic concurrency control without a global counter.

QuasarSTM adopts the per-key $(read\_ts, write\_ts)$ pair (§6) as a tool for *reducing false aborts*. The hard rule is that timestamps *cannot* reorder consensus: the canonical order $\preceq_{\text{can}}$ is supplied by Quasar and is non-negotiable. TicToc-style timestamps only prove that an observed version is serializable *within* the canonical order—they never modify which transaction commits before which.

2.5 GPU-STM Literature: SIMT Livelock and Scalability

Cederman et al. [10] and Holey et al. [13] report two recurrent pathologies on GPU-STM:

1. *SIMT livelock.* Hot-key contention causes warps to repeatedly stomp on each other; non-deterministic victim selection and exponential backoff transplanted from CPU STM do not converge under the lockstep execution model.
2. *Scalability collapse.* A single global version map or hot-key spinlock serialises the entire warp grid, collapsing throughput by orders of magnitude under skewed (Zipfian) access patterns.

QuasarSTM rejects both root causes by construction:

- **Determinism is mandatory.** The contention manager has six fixed policies (§9); there is no randomised victim selection.

- **Bounded repair.** Constants `MAX_FAST_REPAIRS` and `MAX_TOTAL_REPAIRS` (§9) cap repair attempts; transactions that exceed them are promoted to a cold queue or routed through a semantic reducer.
- **Lane refraction.** Hot lanes are isolated by Prism *before* STM sees them (§7); the GPU-STM hot-key pathology never arises in steady state.
- **No global locks.** There is no global STM clock, no global version map mutex, no per-key spinlock, no retry-until-success loop.

2.6 Aria, ESSN, and Multiversion Serialization Graphs

Aria [15] executes a batch against a single snapshot and deterministically commits the maximal serializable subset. ESSN (epoch-serializable safe net) [1] formalises validation against a multiversion serialization graph (MVSG). Both are relevant to QuasarSTM: §14 discusses how the 3.1 evolution adopts Aria-style commit selection and an MVSG-derived validation rule. Within the 3.0 specification, validation is the simpler "visible-version equality under canonical order" form (§5).

3 The Lane Abstraction

The primary innovation of QuasarSTM is replacing the storage key as the unit of independence with the *lane*. A lane is a deterministic projection of the touched state onto a domain that is intuitive to the application, observable to the scheduler, and finer-grained than the contract address yet coarser than the storage slot. Lanes make contention *predictable* before execution.

3.1 Definition

Definition 3.1 (Lane)**.** *A* lane *is the smallest independently schedulable state domain. Each lane has an identifier*

$$lane\_id = H\big(domain,\ contract,\ account,\ asset,\ market,\ nonce\_lane,\ storage\_prefix\big)$$

where H is a 256-bit collision-resistant hash. The components are concrete fields of the touched state; missing components are encoded as a fixed sentinel.

Definition 3.2 (Lane Hint)**.** *A* lane hint *is a tuple $StmLaneHint = (lane\_id,\ access\_kind,\ confidence)$ where $access\_kind \in \{\textsc{Read}, \textsc{Write}, \textsc{Append}, \textsc{Reduce}, \textsc{Unknown}\}$ and $confidence \in [0, 255]$ is the predictor's a priori estimate that the access will occur.*

A transaction's lane hint vector is computed during admission: by static ABI declaration when available, by historical observation of the contract/selector pair otherwise, and by Tier-3 reducer registration for known commutative operations.

3.2 Canonical Lane Examples

3.3 Lane Clocks

Definition 3.3 (Lane Clock)**.** *A* lane clock *is a monotonic counter $StmLaneClock = (version,\ hotness,\ conflict\_$ associated with each $lane\_id$. The `version` component increments on every committed write to the lane; the other components are telemetry.*

The clock is a per-lane shard of the TL2 idea. Lane clocks are stored contiguously in a device-resident hash arena keyed by $lane\_id$, with collision handling by linear probing under $\geq 4\times$ load-factor headroom. There is no global clock *anywhere* in QuasarSTM.

Lane	Definition
sender nonce lane	$H(\text{“nonce”}, \mathit{account\_id}, \mathit{nonce\_idx})$
account balance lane	$H(\text{“acct”}, \mathit{account\_id})$
ERC-20 holder lane	$H(\text{“erc20”}, \mathit{token}, \mathit{holder})$
order-book price-level	$H(\text{“ob”}, \mathit{market\_id}, \mathit{price\_level})$
margin account lane	$H(\text{“margin”}, \mathit{account\_id})$
pool reserve lane	$H(\text{“pool”}, \mathit{pool\_id})$
audit append lane	$H(\text{“audit”}, \mathit{chain\_id}, \mathit{epoch})$
fee accumulator lane	$H(\text{“fee”}, \mathit{token})$

Table 1: Canonical lane definitions for the regulated-DEX workload class. Each lane is a deterministic projection; lane membership is computable without executing the transaction.

3.4 Lane Snapshot

At execution start a transaction T snapshots the current `version` of every lane it touches:

$$\mathit{LaneSnapshot}(T) = \{(L_i, c_i)\}_{i=1}^{m} \quad \text{where} \quad c_i = \mathit{LaneClock}[L_i].\mathit{version}.$$

The snapshot is the precondition for the Tier-1 fast validation (§4): if at validation time $\mathit{LaneClock}[L_i].\mathit{version} = c_i$ for all i, the transaction is committable without further checks.

3.5 Why Lanes (Not Keys) Are the Right Unit

Predictability. Lane membership is computable from the transaction envelope without executing the transaction. A scheduler can therefore partition the ready frontier by lane conflict before any work hits the EVM. Storage keys are not generally known until execution; they are an outcome, not a precondition.

Aggregation. A single hot lane (ERC20.TOTALSUPPLY, AMM reserve, fee counter) often subsumes thousands of distinct storage keys whose semantics commute. Treating these as a single lane and applying a semantic reducer (§4) extracts orders of magnitude more parallelism than per-key MVCC.

Multi-GPU sharding. Cross-GPU coherence on storage keys is intractable at scale; the working set is too fragmented. Lanes give a stable hash domain: $\mathit{gpu\_id} = H(\mathit{lane\_id}) \bmod \mathit{num\_gpus}$ (§11) yields a deterministic, well-balanced sharding plan that survives recompilation.

Telemetry. Hot-lane detection ($\mathit{hotness} = \mathit{reads} + 4{\cdot}\mathit{writes} + 8{\cdot}\mathit{conflicts} + 16{\cdot}\mathit{repairs}$) is naturally per-lane. Per-key hotness aggregates across thousands of keys before producing a useful signal; per-lane hotness is actionable within a single round.

Repair priority. Cold-lane repairs are background work; hot-lane repairs preempt the queue. Without a lane abstraction, the scheduler has no signal to distinguish them.

Remark 3.1 (Lane is necessary, not sufficient)**.** *Lane prediction does not eliminate MVCC. Misprediction must remain* safe*: a transaction whose actual access pattern differs from its hint must still commit a result equivalent to its serial execution under* $\preceq_{\text{can}}$*. The Tier-2 key-MVCC validator (§4) is the safety net. Lanes are how QuasarSTM gets fast on the common case; key-MVCC is how it stays correct on the uncommon case.*

4 Three-Tier Validation

QuasarSTM validates a speculatively executed transaction in three tiers, falling through only when the previous tier cannot prove serializability. The tiers are ordered by cost: Tier 1 touches only lane clocks, Tier 2 walks the read set against per-key MVCC chains, Tier 3 invokes a registered semantic reducer.

4.1 Tier 1: Lane-Clock Fast Validation

Definition 4.1 (Tier-1 validation predicate)**.** *Let T be a speculatively executed transaction with lane snapshot $\mathit{LaneSnapshot}(T) = \{(L_i, c_i)\}_{i=1}^{m}$. Tier-1 validation succeeds for T iff*

$$\forall i \in [m] : \quad \mathit{LaneClock}[L_i].\mathit{version} = c_i.$$

Lemma 4.1 (Tier-1 sufficiency)**.** *If T passes Tier-1 validation under $\preceq_{\text{can}}$, then every read recorded by T observed the visible version under $\preceq_{\text{can}}$, and every write of T is the next visible version on its key.*

Sketch. Lane clocks advance only on committed writes to the lane (§3). The snapshot c_i was taken before T began; the equality at validation implies no committed write to L_i has intervened. Each storage key k touched by T belongs to exactly one lane $L(k)$; if $\mathit{LaneClock}[L(k)]$ is unchanged, the visible version of k under $\preceq_{\text{can}}$ at T's canonical position is unchanged, so T's observed version is correct. □

Tier-1 is the design's fast path. On the disjoint-lane workload (§13 bench A), Tier-1 hit rate exceeds 95%.

4.2 Tier 2: Key-Level MVCC Validation

When a lane clock has advanced, Tier-1 cannot prove the read was unaffected. Tier-2 walks the read set entry-by-entry against the canonical MVCC chain.

Definition 4.2 (Read record)**.** *Each read recorded by T is a tuple $\mathit{StmRead} = (\mathit{tx\_id}, \mathit{key}, \mathit{version\_id}, \mathit{observed}$*

Definition 4.3 (Tier-2 validation predicate)**.** *Tier-2 validation succeeds for T iff for every read record $(\cdot, k, v, \cdot, \cdot)$ recorded by T, the version v is the visible version of k under $\preceq_{\text{can}}$ at T's canonical position (§5, Definition 5.4).*

A Tier-2 mismatch sends T either to Tier-3 (if all mismatched writes are in commutative ops) or directly to the repair queue (§8).

4.3 Tier 3: Semantic Commutativity Validation

Many writes commute deterministically. Treating them as ordinary Set operations is correct but conservative: it produces spurious conflicts whenever two transactions `Add` to the same counter. Tier-3 admits these commits semantically.

Definition 4.4 (Semantic write op)**.** *A write is tagged with a* semantic op *$\mathit{StmWriteOp} \in \{$Set, Add, Sub, Min, Max, Append, BitOr, BalanceDelta, OrderAppend, FeeAccumulate$\}$.*

Definition 4.5 (Commutative on lane)**.** *A pair of semantic ops (o_1, o_2)* commutes on lane L *iff applying o_1 then o_2 to the lane state yields the same post-state as o_2 then o_1, for all valid pre-states.*

The commutativity matrix is fixed by specification:

Definition 4.6 (Tier-3 validation predicate)**.** *Tier-3 validation succeeds for T iff every Tier-2 mismatch is attributable to a write whose semantic op commutes (under Definition 4.5) with every other transaction's mismatching write on the same lane, and the lane has a registered* reducer *that combines the contributions deterministically by canonical index.*

Op pair	Commutes?
Append + Append	yes (deterministic by canonical index)
Add + Add	yes (sum reducer)
FeeAccumulate	yes (sum reducer)
BalanceDelta + BalanceDelta	yes if no overdraft, canonical netting
OrderAppend	yes within batch-auction lane
Min/Max/BitOr	yes (idempotent reducers)
Set + anything	no

Table 2: Semantic commutativity matrix. Tier-3 admits a transaction whose conflicting writes are all in commuting cells.

Reducers. A lane reducer is a per-lane function r_L with type $\mathit{Value} \times \mathit{Op}^* \to \mathit{Value}$ that consumes the lane's pre-state and the canonical-order sequence of contributing ops to produce the post-state. Reducers are pure, deterministic, and gas-metered. They run in the `StmSemanticValidate` workgroup (service ID `0x0432`) ahead of commit.

Why Tier-3 is mandatory for regulated DEX. The order-book `Append`, fee accumulation, balance delta, and audit append lanes dominate exchange workloads. Without Tier-3, these become hot-key conflicts that repeatedly abort. With Tier-3, they collapse to a single deterministic reduction per lane per round. The performance gap is the difference between a fast blockchain and a fast regulated DEX.

4.4 Tier Cascade

Algorithm 1 Validate transaction T

```
if Tier-1 succeeds for T then return LaneValidated
if Tier-2 succeeds for T then return KeyValidated
if Tier-3 succeeds for T then return SemanticallyValidated
return RepairPending
```

Remark 4.1 (Cost monotonicity)**.** *Tier-1 inspects $|m|$ lane clocks (typically $m \leq 4$). Tier-2 walks $|R(T)|$ read records (typically tens). Tier-3 dispatches a reducer on the affected lanes. The cost cascade is exactly inverted from frequency: most transactions stop at Tier 1, which is the cheapest. The pathology Block-STM 2.0 suffered—always running the expensive validator—is eliminated by structure.*

5 Visibility Rule

QuasarSTM serves both Nova (linear consensus order) and Nebula (DAG consensus order) execution modes. The visibility rule is parameterised over a mode-specific *ordering oracle*.

5.1 Ordering Oracle

Definition 5.1 (Ordering oracle)**.** *An ordering oracle is a tuple $\mathit{StmOrderOracle} = (\mathit{mode}, \preceq_{\text{can}}, \mathit{idx})$ where*

- *$\mathit{mode} \in \{\textsc{Nova}, \textsc{Nebula}\}$;*
- *$\preceq_{\text{can}}$ is a total order over committed transactions (Nova) or a strict partial order with deterministic linear extension over the ready cut (Nebula);*
- *idx is a canonical-index function from transaction identifiers to $\mathbb{N}$ that respects $\preceq_{\text{can}}$.*

The oracle is supplied by Quasar before the round begins; it is fixed for the duration of the round.

The oracle is the only consensus-order surface the STM engine consumes. It abstracts the linear/DAG distinction at a single point.

5.2 Nova Visibility

Definition 5.2 (Nova visible version)**.** *Let T_i be a transaction with canonical index i and let k be a storage key. Under Nova,*

$$\mathit{visible\_version}(T_i, k) = v(\,T_j,\, k\,) \quad \text{where} \quad j = \max\{\, j' < i \;:\; T_{j'} \text{ wrote } k,\ \text{and } v(T_{j'}, k) \text{ is valid} \,\}.$$

If no such j exists, the visible version is the parent-state baseline of k.

A version $v(T_j, k)$ is *valid* if the latest incarnation of T_j that wrote k has not been invalidated by a subsequent repair (§8, repair monotonicity). Speculative versions of later incarnations supersede earlier ones; only the topmost valid incarnation participates in visibility.

5.3 Nebula Visibility

In Nebula, transactions are vertices in a causal DAG; the canonical order is a deterministic linear extension over the Prism cut.

Definition 5.3 (Nebula visible version)**.** *Let v_i be a vertex (transaction) in a Nebula round and let k be a storage key. Let $\mathit{Pred}^*(v_i)$ be the set of vertices causally before v_i in the DAG (transitive predecessors under the happens-before relation). Under Nebula,*

$$\mathit{visible\_version}(v_i, k) = v(\,v_j,\, k\,) \quad \text{where} \quad v_j = \operatorname{argmax}_{\preceq_{\text{can}}}\{\, v_{j'} \in \mathit{Pred}^*(v_i) \;:\; v_{j'} \text{ wrote } k,\ v(v_{j'}, k) \text{ valid} \,\}.$$

The argmax is taken under the deterministic linear extension supplied by the ordering oracle, which decides ties between concurrent (DAG- incomparable) writers.

Lemma 5.1 (Tie-break determinism)**.** *For any two concurrent vertices $v_a, v_b \in \mathit{Pred}^*(v_i)$ with $v_a \parallel v_b$ in the DAG, the ordering oracle's linear extension yields a fixed total order independent of execution ordering, scheduling, or non-determinism on the GPU.*

Sketch. The Nebula linear extension is computed deterministically from the DAG structure (Photon proposal hashes, Wave finalization indices) together with a fixed tie-break rule on (round, vertex hash). None of those inputs depend on execution. The lemma is structural. □

5.4 Mode-Independent Statement

Definition 5.4 (Visible version)**.** *For an ordering oracle $\mathcal{O}$ and a transaction-key pair (T, k), the visible version $\mathit{visible\_version}_{\mathcal{O}}(T, k)$ is given by Definition 5.2 when $\mathcal{O}.\mathit{mode} = \textsc{Nova}$ and by Definition 5.3 when $\mathcal{O}.\mathit{mode} = \textsc{Nebula}$.*

The Tier-2 validation predicate (Definition 4.3) refers to exactly this $\mathit{visible\_version}_{\mathcal{O}}$.

5.5 Canonical Version Order

Definition 5.5 (Canonical version order)**.** *Versions are totally ordered by the lexicographic key $(\mathit{key}, \mathit{tx\_index}, \mathit{incarnation})$. The visible-chain on each key, projected from the global version arena, respects this order under collision-free indexing.*

Remark 5.1 (No race-dependent visibility)**.** *Two GPU threads writing speculative versions for the same key under the same canonical index but different incarnations produce a deterministically sorted visible chain. There is no tie-break by write timestamp, no first-writer-wins, no last-writer-wins. The ordering is structural.*

6 TicToc-Style Timestamps Without Breaking Determinism

QuasarSTM borrows TicToc's per-data-item timestamp pair to reduce false aborts on the Tier-2 path. The hard rule (§2) is that timestamps cannot reorder consensus—they only certify that an observed version is serializable within the canonical order $\preceq_{\text{can}}$.

6.1 Per-Key Timestamps

Definition 6.1 (Per-key timestamps)**.** *Each key header carries* $(read\_ts,\ write\_ts) \in \mathbb{N}^2$ *where*

- $read\_ts(k) = \max\{\ ts(T) : T \text{ has read } k\ \}$*;*
- $write\_ts(k) = ts(T)$ *for the most recent committed writer* T *of* k*.*

Both fields are stored in the device-resident `StmKeyHeader`*.*

6.2 Per-Transaction Timestamps

Definition 6.2 (Per-transaction timestamps)**.** *For a transaction* T *with read footprint* $R(T)$ *and write footprint* $W(T)$*,*

$$T.read\_ts\_min = \max\big\{\ write\_ts(k)\ :\ k \in R(T)\ \big\}$$

$$T.write\_ts = \max\big\{\ read\_ts(k)\ :\ k \in W(T)\ \big\} + 1.$$

A transaction passes the TicToc test *if there exists* $ts(T) \in [T.read\_ts\_min,\ T.write\_ts]$ *that is consistent with the canonical position* $idx(T)$ *supplied by the ordering oracle.*

6.3 Reduction of False Aborts

Lemma 6.1 (TicToc admits more commits than version-equality)**.** *Let* T *be a transaction whose Tier-2 read-set check fails (some recorded* $version\_id$ *is no longer the* $visible\_version_{\mathcal{O}}$*). If* T *passes the TicToc test (Definition 6.2) and the canonical linearisation of the new visible version places it in a serial position equivalent to* T*'s observation, then* T *commits.*

Sketch. A version-equality failure is a sufficient but not necessary condition for an abort. If T's read footprint is satisfied by some $ts(T)$ that respects the canonical-order envelope, then the serial schedule $T_1 \preceq_{\text{can}} \cdots \preceq_{\text{can}} T \preceq \cdots$ is equivalent to a schedule in which T observes the new visible version, because the timestamp interval certifies no intervening write conflicts with T's read. □

6.4 Determinism Guard

Invariant 6.1 (TicToc determinism)**.** *For all transactions* T *and all valid timestamp intervals* $[read\_ts\_min,\ write\_ts]$*,*

$$ts(T)\ \textit{admissible} \implies ts(T)\ \textit{consistent with}\ \preceq_{\text{can}}.$$

That is: no choice of $ts(T)$ *may move* T *before any* T_j *with* $T_j \preceq_{\text{can}} T$ *in the committed result.*

The implementation enforces Invariant 6.1 at the validator: when the TicToc test admits an interval, the chosen $ts(T)$ is the unique minimum that satisfies both the interval and $\preceq_{\text{can}}$. There is no degree of freedom for the GPU to choose differently across runs.

Remark 6.1 (No reordering)**.** *TicToc* cannot *produce a committed schedule that violates* $\preceq_{\text{can}}$*. It only avoids aborting transactions whose observations are still serializable under that order. This is the defining departure from canonical TicToc, which derives the order* from *the timestamps. In QuasarSTM the order comes from Quasar; the timestamps only justify the absence of an abort.*

Field	Size
`StmKeyHeader.read_ts`	8 B
`StmKeyHeader.write_ts`	8 B
`StmRead.observed_write_ts`	8 B
`StmRead.observed_lane_clock`	8 B

Table 3: Timestamp storage overhead per key and per read record. The two key-header fields fit in the slack of the existing 32-byte slot; the per-read fields cost 16 B each.

6.5 Storage Cost

7 Prism Refraction

Prism is QuasarSTM's primary contention-prevention layer. Block-STM 2.0 let conflicts happen and repaired them; Prism in 3.0 partitions the ready frontier so that conflicting transactions execute in *different* slices and the STM rarely sees the same hot conflict twice.

7.1 Lane Conflict Matrix

Definition 7.1 (Lane conflict matrix)**.** *Given a ready frontier $F = \{T_1, \ldots, T_N\}$ with lane hint vectors $LH(T_i)$, the lane conflict matrix $M \in \{0,1\}^{N \times N}$ is defined by*

$$M_{ij} = \begin{cases} 1 & \text{if } \exists L \in LH(T_i) \cap LH(T_j) \text{ s.t. at least one access is } \textsc{Write}, \\ 0 & \text{otherwise.} \end{cases}$$

M is symmetric; $M_{ii} = 0$.

The matrix is computed in linear-arithmetic time on the GPU using hashed lane bucket counts; full pairwise comparison is unnecessary. The construction is deterministic.

7.2 Slice Partition

Definition 7.2 (Refraction partition)**.** *A refraction of F is a partition $F = S_1 \sqcup \cdots \sqcup S_q$ such that every slice S_p falls into one of three classes:*

1. *DisjointWrite: $\forall T_i, T_j \in S_p,\ M_{ij} = 0$.*
2. *ReadOnly: every $T_i \in S_p$ has only Read lane hints (and no writes will alter lane clocks).*
3. *HotIsolated: $S_p = \{T : L^* \in LH(T)\}$ for a single hot lane L^*; the slice is processed under a Tier-3 reducer or a serialised lane mode.*

Algorithm 2 Compute refraction of frontier F

1: Bucket transactions by lane hint into a per-lane occupancy table.
2: Mark hot lanes L^* where occupancy exceeds the hot-lane threshold θ_{hot} or where $LaneClock[L^*].hotness$ exceeds θ_{hotness}.
3: **for each** hot lane L^*: emit HotIsolated slice $S_p \leftarrow \{T : L^* \in LH(T)\}$ and remove these transactions from F.
4: Greedy-pack remaining transactions into DisjointWrite slices using a deterministic colouring of the (residual) lane conflict graph.
5: Emit a single ReadOnly slice for purely-read transactions.
6: **return** $\{S_1, \ldots, S_q\}$.

Lemma 7.1 (Refraction safety)**.** *For any frontier F and any refraction $\{S_1, \ldots, S_q\}$ satisfying Definition 7.2, executing each slice S_p in parallel and serialising slices in the canonical order respects $\preceq_{\text{can}}$.*

Sketch. Within DisjointWrite, no two transactions touch a common write lane, so no MVCC conflict is possible; any read-set under unchanged lane clocks satisfies Tier-1. Within Read-Only, no writes alter visibility, and the slice's transactions all observe the same pre-state. Within HotIsolated, the registered Tier-3 reducer or serialised-lane policy produces the canonical-order outcome. Across slices, the canonical order is respected by sequential slice dispatch. □

7.3 Refraction Determinism

Invariant 7.1 (Refraction determinism)**.** *For any frontier F and any pair of executions of Algorithm 2 on the same F and the same lane hint vectors, the produced partition is bit-identical (up to slice ordering, which is itself fixed by canonical-index lexicography).*

This is mandatory: refraction sits on the round-result hash path. A non-deterministic refraction would propagate to `execution_root` and break consensus.

7.4 Why Refraction Is the Performance Primitive

Same hot conflict twice. Without refraction, a frontier with 64 transactions all writing to the AMM reserve produces $\binom{64}{2} = 2016$ pairwise conflicts on the same lane. The validator detects them, the repair queue absorbs them, and incarnation counters climb. With refraction, all 64 transactions land in a single HotIsolated slice and are processed by a registered AMM reducer that yields exactly one deterministic post-state.

Lane-fast hit rate. Refraction is the precondition for the Tier-1 hit-rate target ($> 95\%$ on disjoint-lane workloads, §13). Without partitioning, any single hot transaction in a frontier disturbs every co-resident transaction's lane snapshot.

Telemetry feedback loop. Hot-lane detection feeds the next round's refraction. A lane that caused $> \theta_{\text{hot}}$ conflicts in round r is promoted to HotIsolated for round $r + 1$ *before* the conflicts recur. The control loop closes within one Quasar round.

8 Incremental Repair via Fiber Checkpoints

Block-STM's classical repair is full transaction re-execution at the next incarnation. For complex EVM flows (multi-hop swap routers, compliance precompiles, batched settlement) this is wasteful: the invalidated read often occurs late, and re-running prefixes of pure arithmetic and warm storage accesses dominates wall time.

QuasarSTM repairs against *fiber checkpoints* taken at known boundaries during execution. A repair rolls back to the most recent checkpoint preceding the invalidated read, retains the warm bytecode, calldata, journal prefix, and earlier validated reads, and resumes the fiber from that point with the new visible version.

8.1 Checkpoint Boundaries

The Metal/CUDA EVM fiber takes checkpoints at six structural boundaries:

1. Call boundaries (`CALL`, `STATICCALL`, `DELEGATECALL`, `CALLCODE`; deferred to v0.40+).
2. Sload / Sstore boundaries.
3. Precompile boundaries (entry and exit of any registered precompile).
4. Basic-block budget boundaries (every B opcodes, where B is a per-fiber budget chosen to bound checkpoint memory).

5. DEX semantic-operation boundaries (order append, cancel, match, fee accumulate, balance settle—the Tier-3 reducer-eligible operations).
6. Cold-state resume boundaries (when a fiber resumes from a `StateResp`, the resume point is implicitly a checkpoint).

Definition 8.1 (Fiber checkpoint)**.** *A fiber checkpoint is a tuple*

$$\mathit{EvmFiberCheckpoint} = (\mathit{tx\_id}, \mathit{pc}, \mathit{frame\_offset}, \mathit{stack\_offset}, \mathit{memory\_offset}, \mathit{journal\_offset}, \mathit{read\_set\_co}$$

where the offsets index into the fiber's per-round arenas. A checkpoint is a constant-size record (typically 32 B) that does not copy memory; it pins arena prefixes by index.

8.2 Repair Policy

Algorithm 3 Repair transaction T after invalidated read at canonical position p

$K \leftarrow$ the most recent checkpoint of T with checkpoint position $\leq p$.
if K exists **then**
  Roll back fiber: restore pc, $\mathit{stack\_offset}$, $\mathit{memory\_offset}$, $\mathit{journal\_offset}$, $\mathit{read\_set\_count}$ to K's recorded values.
  Discard read records and write records appended after $K.\mathit{read\_set\_count}$.
  Increment $T.\mathit{incarnation}$.
  Re-enter EXECUTING state at $K.pc$, observing the current visible version.
else
  Increment $T.\mathit{incarnation}$.
  Re-enter EXECUTING from start (full re-exec).

8.3 Repair Telemetry

Counter	Definition
`checkpoint_rollback_count`	repairs that restored to a checkpoint
`full_reexec_count`	repairs that ran from start
`rollback_saved_gas`	gas not re-executed under rollback
`rollback_saved_wave_ticks`	ticks not re-executed under rollback

Table 4: Repair telemetry exported in `QuasarStmTelemetry`.

8.4 Cost Reduction

Lemma 8.1 (Repair cost upper bound)**.** *Let T be a transaction with fiber checkpoints at gas-cumulative positions $0 = g_0 < g_1 < \cdots < g_K \leq G(T)$, where $G(T)$ is T's total gas. Let g^* be the gas-cumulative position of the invalidated read. Then the gas re-executed under Algorithm 3 is bounded above by $G(T) - g_j$ where $g_j = \max\{g_i : g_i \leq g^*\}$.*

Sketch. The rollback restores state at g_j; execution proceeds from g_j to the end. By construction $g_j \leq g^* \leq G(T)$, so the cost is at most $G(T) - g_j$. □

Remark 8.1 (5–30× on routers)**.** *On router and DEX-settlement workloads (bench E, §13), measured repair cost falls 5–30× relative to full re-execution. This is the fattest single optimisation in the repair path; without it, hot-frontier rounds suffer substantial repair amplification.*

9 Deterministic Contention Manager

GPU-STM literature [10, 13] catalogues SIMT livelock and unbounded retry as the dominant failure modes of naive ports. QuasarSTM's contention manager is deterministic by construction: there is no randomised victim selection, no spinlock, no retry-until-success loop.

9.1 Conflict Policies

Definition 9.1 (Conflict policy set)**.** *The contention manager has six fixed policies:*

$$\mathit{StmConflictPolicy} \in \{\textsc{EarlierWins}, \textsc{HotLaneSerialize}, \textsc{RefractionSplit}, \textsc{IncarnationBackoff}, \textsc{Se}$$

Policies are applied in a fixed cascade.

Algorithm 4 Contention manager: dispatch policy for conflict C between T_a and T_b

1: (T_a, T_b) ordered so $idx(T_a) < idx(T_b)$.
2: **Apply EarlierWins**: T_a keeps its speculative result; T_b enters RepairPending.
3: **if** same lane L has produced $> \theta_{\text{hot}}$ conflicts in this round **then** apply HotLaneSerialize: lane L promoted to hot-isolated; subsequent transactions on L run in canonical order.
4: **if** a registered Tier-3 reducer exists for L **then** apply SemanticReducer: route T_b through the reducer instead of re-executing.
5: **if** frontier-wide conflict density exceeds θ_{frontier} **then** apply RefractionSplit: future Prism cuts narrow.
6: **if** $T_b.\mathit{incarnation} >$ `MAX_FAST_REPAIRS` **then** apply IncarnationBackoff: T_b moved to a slower service queue with extended budget.
7: **if** $T_b.\mathit{incarnation} >$ `MAX_TOTAL_REPAIRS` **then** apply ColdQueueDemotion: T_b removed from this round and deferred to the next.

Lemma 9.1 (Determinism of contention manager)**.** *Algorithm 4 produces the same policy decision under any two GPU executions on the same input frontier and the same lane state. There is no degree of freedom for tie-breaking beyond canonical-index ordering.*

Sketch. Every branch condition is a deterministic function of $(T_a, T_b, \mathit{LaneClock}, \mathit{TelemetryCounters})$, all of which are device-resident structures updated by deterministic write paths. The cascade is total: at most one policy fires for any given (T_a, T_b, L) tuple. The choice of T_b as the victim is forced by canonical-index ordering, not by GPU thread ID. □

9.2 Hard Limits

Definition 9.2 (Repair hard limits)**.** *The contention manager enforces three constants:*

$$\begin{aligned} \texttt{MAX\_FAST\_REPAIRS} &= 3 \\ \texttt{MAX\_TOTAL\_REPAIRS} &= 8 \\ \texttt{HOT\_LANE\_CONFLICT\_THRESHOLD} &= 16 \end{aligned}$$

Lemma 9.2 (No unbounded retry)**.** *For any transaction T and any sequence of conflicts involving T, the number of repairs T undergoes within a single Quasar round is bounded by* `MAX_TOTAL_REPAIRS`*. After this point T is demoted to the cold queue and runs in the next round.*

Proof. Algorithm 4 step 7 applies ColdQueueDemotion when $T_b.\mathit{incarnation} >$ `MAX_TOTAL_REPAIRS`. The incarnation counter is monotonically increasing under repair monotonicity (§12, Invariant 12.1), so the predicate eventually fires within finitely many repair rounds. □

9.3 Hot-Lane Modes

Definition 9.3 (Hot-lane mode)**.** *A lane promoted by* HotLaneSerialize *or by hotness threshold acquires one of four modes:*

$$\mathit{HotLaneMode} \in \{\textsc{Serialize}, \textsc{Reduce}, \textsc{Precompile}, \textsc{Defer}\}.$$

Hot lane	Mode
erc20.totalSupply	Reduce or Serialize
AMM reserves	Precompile (batch auction)
order-book price level	Reduce (Append + match)
fee counter	Reduce (sum reducer)
sender nonce	Serialize per sender
compliance identity	Serialize or snapshot

Table 5: Hot-lane mode assignments for the canonical regulated-DEX contract set.

Hot lanes are not pathologies; they are where application economics live. QuasarSTM 3.0 makes them explicit primitives.

9.4 Forbidden Patterns

The QuasarSTM specification prohibits the patterns documented as fatal in the GPU-STM literature:

- one global STM clock;
- one global version map lock;
- retry-until-success GPU loops;
- nondeterministic conflict victim selection;
- opcode-level STM (QuasarSTM is transaction-level);
- CPU-side conflict manager;
- host-side repair scheduler;
- per-version `malloc` (the per-round arena is mandatory);
- global hot-key spinlocks.

These either break determinism or collapse GPU throughput. Compliance with this list is a build-time invariant: the round path on Metal and CUDA is statically inspected for these patterns.

10 Commit Horizon

Per-transaction commit imposes a fixed cost (root accumulation, receipt hashing, telemetry counters) on every transaction. For a workload of N transactions and per-tx commit cost c, the total overhead is $\Theta(Nc)$. A commit horizon amortises this cost over contiguous valid prefixes (Nova) or valid causal cuts (Nebula), collapsing the overhead to $\Theta(\mathit{horizon\_count} \cdot c)$ where $\mathit{horizon\_count} \ll N$.

10.1 Definition

Definition 10.1 (Commit horizon)**.** *A commit horizon is a tuple*

$$\mathit{StmCommitHorizon} = (\mathit{start\_index}, \mathit{end\_index}, \mathit{horizon\_root}, \mathit{tx\_count}).$$

A horizon is valid *iff:*

1. *every transaction T with $start\_index \le idx(T) \le end\_index$ has reached state* Committable *or beyond;*
2. *no unresolved earlier dependency exists (no transaction with $idx < end\_index$ remains in* Executing, Refracted, *or* RepairPending*);*
3. *no* SuspendedState *request blocks visibility within the horizon;*
4. *all hot-lane reducers touching keys in the horizon's write set have finalised.*

10.2 Mode-Specific Horizon Shape

Definition 10.2 (Nova horizon)**.** *Under Nova, a horizon $[start\_index, end\_index]$ is a contiguous prefix of the canonical linear order. The next horizon begins at $end\_index + 1$.*

Definition 10.3 (Nebula horizon)**.** *Under Nebula, a horizon is a valid causal cut: a downward-closed subset H of the round's DAG (i.e., $v \in H$ implies $Pred^*(v) \subseteq H$). The Photon / Wave finalisation indices supply the cut boundary; the Flare layer furnishes the deterministic linear extension that the horizon_root commits to.*

Lemma 10.1 (Horizon completeness)**.** *For any Quasar round, every committed transaction belongs to exactly one commit horizon.*

Proof. Horizons are produced by the `StmCommitHorizon` service in canonical order; each transaction enters exactly one horizon when the predicate of Definition 10.1 fires. Within a round there is no overlap and no gap. □

10.3 Root Construction over Horizons

The *horizon_root* is computed by the `StmCommit` service as a Keccak-f[1600] hash chain over the canonical-order sequence of per-transaction receipt hashes within the horizon. The chain seed is the previous horizon's root; the final horizon's output feeds directly into `execution_root` on the round result.

Definition 10.4 (Round result fields fed by horizons)**.** *The QuasarRoundResult derives the following fields from horizon material:*

- `block_hash`*: finalisation digest over horizon roots.*
- `state_root`*: accumulated commit chain across horizons.*
- `receipts_root`*: per-tx receipt hash chain rooted at horizon boundaries.*
- `execution_root`*: Block-STM trace commitment over horizons.*
- `mode_root`*: Nova/Nebula-specific structural commitment.*

10.4 Horizon Granularity

Remark 10.1 (Empirical horizon count)**.** *On the 1024-tx end-to-end stress (§13), a typical round produces 8–16 horizons. Per-tx commit cost amortises to roughly $1024/12 \approx 85$ transactions per horizon, an $85\times$ reduction in per-tx commit overhead relative to the 2.0 baseline.*

The horizon count is workload-dependent: high-conflict workloads produce more horizons (each cut waits for repair to settle) while disjoint-lane workloads produce few.

10.5 Horizon Safety

Invariant 10.1 (Horizon safety)**.** *No transaction T with an unresolved dependency may enter any commit horizon. Formally, if there exists T' with $T' \preceq_{\text{can}} T$ and T' is in a non-terminal state (*Executing, Refracted, RepairPending, SuspendedState, Repairing*), then T is excluded from the current horizon.*

This is the precondition that makes the horizon root meaningful: the root commits only to a state in which all canonical predecessors are stable.

11 Garbage Collection and Multi-GPU Sharding

QuasarSTM allocates speculative versions in a per-round arena. If versions are not aggressively reclaimed the arena saturates within a few rounds. Multi-GPU operation imposes the additional constraint that lane state must shard cleanly across devices without cross-GPU hot keys.

11.1 Epoch / Horizon GC

Definition 11.1 (Safe GC point)**.** *The safe GC point of a round is*

$$\begin{aligned} \mathit{safe\_gc\_point} = \min\big(\, & \mathit{lowest\_active\_tx\_index}, \\ & \mathit{lowest\_repair\_dependency}, \\ & \mathit{current\_commit\_horizon}, \\ & \mathit{lowest\_live\_checkpoint\_reference}\,\big). \end{aligned}$$

Definition 11.2 (GC reclamation)**.** *The GC service reclaims, for any record r with canonical position p_r:*

- *invalid incarnations of r where $p_r < \mathit{safe\_gc\_point}$;*
- *superseded speculative versions whose canonical successor has committed;*
- *checkpoint memory not referenced by any active repair;*
- *read records for committed transactions.*

A dedicated workgroup (`ServiceId :: StmGC = 0x0451`) runs the reclaimer concurrently with execution. The arena is multi-producer / single- consumer; GC never blocks the round path.

Counter	Definition
`versions_allocated`	total speculative versions allocated this round
`versions_reclaimed`	versions reclaimed by GC
`checkpoint_bytes_live`	checkpoint memory currently pinned
`mvcc_arena_pressure`	arena occupancy ratio

Table 6: GC telemetry fields exported in `QuasarStmTelemetry`.

11.2 Lane-Keyed Multi-GPU Sharding

Definition 11.3 (Lane sharding function)**.** *For a deployment with G GPUs, the lane sharding function is*

$$\mathit{gpu\_id}(L) = H(\mathit{lane\_id}_L) \bmod G.$$

The map is fixed for the duration of the round.

Definition 11.4 (Local vs distributed transaction)**.** *A transaction T is* local *iff $\{\mathit{gpu\_id}(L) : L \in LH(T)\}$ is a singleton. Otherwise T is* distributed.

11.3 Distributed Transaction Protocol

Algorithm 5 Distributed transaction T touching GPUs $G_1, \ldots, G_k$

1: Each G_i executes the local fragment of T on its lanes, producing a per-GPU fragment root ρ_i.
2: Each G_i validates the local lane clocks ($LaneClock[L].version$) against T's snapshot.
3: Fragment validation results are aggregated under a deterministic reduction (e.g., AND of per-GPU validity bits) at the round leader.
4: **if** all fragments valid **then** commit; else enter coordinated repair (each G_i rolls back its fragment to the latest local checkpoint).

Lemma 11.1 (Distributed atomicity)**.** *For any distributed transaction T, T commits on all involved GPUs or on none.*

Sketch. The aggregator publishes a single commit decision derived from the deterministic reduction of per-GPU validity. Each GPU acts on the aggregated decision; per-GPU local commits are gated on the aggregated bit. There is no path to partial commit. □

11.4 Cross-GPU Invariants

Invariant 11.1 (Cross-GPU lane consistency)**.** *For any lane L and any GPU G_i with $gpu\_id(L) = i$: the canonical writer of L at any point is recorded in G_i's lane-clock arena and nowhere else.*

Invariant 11.2 (Cross-backend equivalence)**.** *For any deterministic round and any pair of backends in* {Metal, CUDA, CPU reference}*, the produced* `state_root`*,* `receipts_root`*,* `execution_root`*, and* `mode_root` *are bit-identical.*

The CPU reference (`evmgpu` Go) is the differential-fuzzing oracle for Metal and CUDA; structural divergence is a build breaker.

12 Correctness Invariants

QuasarSTM is specified by four invariants. Determinism and ordered serializability are the safety properties; repair monotonicity and horizon safety are the structural properties that make the safety properties achievable on a GPU.

12.1 Determinism

Theorem 12.1 (Determinism)**.** *For any input $(D, txs, parent\_state)$ where D is the round descriptor, txs is the canonical-order transaction sequence, and parent_state is the parent state root, two executions of the QuasarSTM round on any pair of backends in* {Metal, CUDA, CPU} *produce bit-identical state_root, receipts_root, execution_root, and mode_root.*

Sketch. By induction on round structure. Refraction is deterministic (Invariant 7.1). Lane and key-version data structures are open-addressed with stable insertion order keyed by canonical index. Tier-1, Tier-2, Tier-3 validation predicates are pure functions of structural data (Definitions 4.1, 4.3, 4.6). The contention manager is deterministic (Lemma 9.1). TicToc determinism is enforced (Invariant 6.1). Horizon construction is canonical-order (Lemma 10.1). Each link in the round-result chain is therefore a deterministic function of its predecessors; the composition is deterministic. The cross-backend invariant (Invariant 11.2) is verified by differential fuzzing against the CPU reference. □

12.2 Ordered Serializability

Theorem 12.2 (Ordered serializability)**.** *For any committed result of a QuasarSTM round under ordering oracle $\mathcal{O}$, there exists a serial schedule S over the committed transactions, equivalent to the committed result, that respects $\preceq_{\text{can}}$.*

Sketch. Each committed transaction passed Tier-1, Tier-2, or Tier-3 validation. By Lemma 4.1, Tier-1 commits observed visible versions. By Definition 4.3, Tier-2 commits did the same. Tier-3 commits respect $\preceq_{\text{can}}$ through deterministic reducer composition (the reducer consumes ops in canonical-index order). TicToc relaxation (Lemma 6.1) admits commits whose observations are *still* serializable under $\preceq_{\text{can}}$, never otherwise (Invariant 6.1). The serial schedule S is the canonical-index ordering of the committed set, augmented with reducer effects in their canonical order; equivalence to the committed result follows from per-tier serializability. □

12.3 Repair Monotonicity

Theorem 12.3 (Repair monotonicity)**.** *Let T be any transaction and let $\iota_1 < \iota_2 < \cdots$ be its incarnation numbers across repairs in a single round. Then:*

1. *Each repair strictly increases the incarnation: $\iota_{i+1} = \iota_i + 1$.*
2. *No incarnation ι_j with $j <$ final commits.*
3. *Only the final incarnation contributes to the round result.*

Sketch. (1) Algorithm 3 step 5 (and step 9 in the full-reexec branch) increments the incarnation counter. (2) The validator discards records tagged with non-final incarnations; speculative versions written by superseded incarnations are GC'd (Definition 11.2). (3) The visible-version chain indexes by $(key, tx\_index, incarnation)$; at commit time only the topmost valid incarnation is exposed. □

Invariant 12.1 (Repair monotonicity invariant)**.** *The incarnation counter on any `StmTxn` record is monotonically non-decreasing throughout a round; an invalidated incarnation cannot be reused.*

12.4 Horizon Safety

Theorem 12.4 (Horizon safety)**.** *For any commit horizon h produced by the round, the canonical prefix (Nova) or causal cut (Nebula) of h contains only transactions in terminal state, and the horizon_root commits to a state in which all canonical predecessors are stable.*

Sketch. Definition 10.1 requires every transaction in the horizon to be COMMITTABLE or beyond and forbids unresolved earlier dependencies. Invariant 10.1 is the validator-enforced predicate. Lemma 10.1 ensures every committed transaction belongs to exactly one horizon in canonical order, so the chain $\rho_0 \to \rho_1 \to \cdots$ of horizon roots is a deterministic commitment over the round. □

12.5 Test-Enforced Round Invariants

In addition to the four headline theorems, the test surface (§13) enforces seven implementation invariants:

1. Same input + same parent state + same mode ⇒ same roots (Theorem 12.1).
2. Committed result equals canonical Nova order or deterministic Nebula causal order (Theorem 12.2).
3. Host cannot choose validate / repair / commit order (host = doorbell + I/O).
4. Every read observes the correct visible version (Definition 5.4).
5. Each repair increments incarnation; invalid incarnations cannot commit (Theorem 12.3).
6. No transaction beyond unresolved dependencies enters a commit horizon (Theorem 12.4).
7. Metal ≡ CUDA ≡ CPU reference for all roots and conflict/repair behaviour (Invariant 11.2).

13 Performance

All measurements were taken on commodity hardware (Apple M1 Max, 10 CPU cores, 32 GPU cores, 64 GB unified memory) under macOS, with the Lux `cevm` substrate at the QuasarSTM 3.0 spec-freeze revision (December 15, 2025). The CUDA backend was sanity-checked on an NVIDIA H100 (80 GB) with identical structural parity.

13.1 End-to-End Stress

Bench	Tx count	Wave ticks	Wall time
`single_tx_real_roots`	1	1	< 2 ms
`multi_tx_counters`	128	2	~ 18 ms
`bounded_backpressure`	1024	32	~ 150 ms
`end_to_end_stress`	1024	8	~ 150 ms

Table 7: End-to-end wall time for representative round shapes on M1 Max. `end_to_end_stress` is the canonical 8-tick 1024-tx benchmark cited in the abstract.

13.2 Hot-Key Contention (Textbook Block-STM)

Workload	Conflicts	Repairs	Commits
`block_stm_independent_txs` (64 disjoint keys)	0	0	64
`block_stm_conflict_repair` (16 same-key)	120	120	16

Table 8: Block-STM behaviour under disjoint-key and same-key contention. The 16 same-key transactions produce exactly $\binom{16}{2} = 120$ pairwise conflicts and 120 repairs, converging to 16 commits. This matches the Aptos [12] reference behaviour, on GPU.

Remark 13.1 (Why the conflict count is exact)**.** *The pairwise count* 120 *is structural: under canonical-index ordering, transaction T_j at position j conflicts with each of $T_0, \ldots, T_{j-1}$ on the shared key. Summing over j from* 1 *to* 15 *yields $\sum_{j=1}^{15} j = 120$. The value is the same on Metal and CUDA and on the CPU reference, by Invariant 11.2.*

13.3 Repair Amplification Targets

Workload	Target *repair_amplification*
disjoint-lane DEX	< 1.01
balanced-lane DEX	< 1.05
heavy-contention DEX	< 2.00, controlled degradation
adversarial Zipfian	bounded by `MAX_TOTAL_REPAIRS`

Table 9: Production targets for repair amplification (*repair_amplification* = *repair_count*/*committed_count*). The headline metric exported on every `QuasarRoundResult`.

If *repair_amplification* > 1.05 for sustained periods, Prism refraction needs tuning: the workload is producing predictable hot lanes that should have been isolated upstream.

Bench	Workload	Target
A	1 M / 10 M / 100 M disjoint-lane transfers	lane-fast (Tier-1) hit $> 95\%$
B	Zipfian key sampling	hot-lane detection + bounded repair amplification
C	AMM hotspot (many swaps, one pool)	generic STM degrades; semantic / precompile path survives
D	Order-book append (many appends, one market)	APPEND reducer + deterministic batch matching
E	Router (long EVM fibers with late `SLOAD` conflict)	checkpoint rollback beats full re-exec by $5{-}30\times$
F	Cold-state suspension (forced page misses)	suspended fibers do not stall the wave scheduler
G	Nebula frontier (DAG antichains of varying conflict density)	Prism refraction reduces `repair_count`

Table 10: LP-010 conformance benchmark ladder. Each row is a required test on the round path; failure on any row is a specification violation.

13.4 Conformance Bench Ladder

13.5 Per-Stage Telemetry on the Stress Round

Counter	Value
`tx_count`	1024
`committed_count`	1024
`conflict_count`	0 (disjoint workload)
`repair_count`	0
`lane_fast_valid_count` (Tier-1)	> 970
`key_valid_count` (Tier-2)	< 55
`semantic_valid_count` (Tier-3)	0
`commit_horizon_count`	8–16
`mvcc_versions_allocated`	$\sim 3.2\times10^3$
`mvcc_versions_reclaimed`	$\sim 3.2\times10^3$
`mvcc_arena_pressure`	< 0.05

Table 11: Telemetry from the 1024-tx end-to-end stress round. Tier-1 absorbs $> 95\%$ of validation; the arena pressure stays under 5% thanks to per-horizon GC.

13.6 Service-Level Latency

13.7 Cross-Backend Determinism

The Metal and CUDA backends share the layout types (`quasar_gpu_layout.hpp`) byte-for-byte; structural drift is caught by `static_assert` at compile time. The CPU reference (`evmgpu` Go) is the differential-fuzzing oracle for both GPU backends. The 19-binary GPU test surface (parity, modes, host-bridge, pipeline, unified, gpu-state, opcodes, dispatch, refund, access-list, stack-depth, storage-overflow, ...) passes on M1 Max; CUDA parity on H100 is verified at v0.41.

14 Related Work

Block-STM (Aptos / Diem). Block-STM [12] introduced ordered speculative execution against a known canonical order. Its central contribution is treating the canonical order as a val-

Service (`ServiceId`)	**Per-tick budget**
`StmPredict` (`0x0410`)	$\sim 1\,\mu$s
`StmRefract` (`0x0411`)	$\sim 8\,\mu$s
`EvmFiberExec` (`0x0420`)	wave-tick dominant
`StmLaneValidate` (`0x0430`)	$\sim 4\,\mu$s per slice
`StmKeyValidate` (`0x0431`)	$\sim 15\,\mu$s per slice
`StmSemanticValidate` (`0x0432`)	per-reducer
`StmRepairSelect` (`0x0440`)	$\sim 2\,\mu$s
`StmRepairExec` (`0x0441`)	per-tx
`StmCommitHorizon` (`0x0450`)	$\sim 3\,\mu$s
`StmGC` (`0x0451`)	background

Table 12: Per-service GPU wave-tick budget on M1 Max. Validation splits cleanly across services; the wave-tick scheduler [17] interleaves them within a single bounded kernel re-launch.

idation primitive rather than a serialisation bottleneck. QuasarSTM 3.0 inherits the speculative-execute / validate / repair structure and extends it with lane abstraction, three-tier validation, semantic reducers, checkpoint- based incremental repair, deterministic contention management, commit horizons, and multi-GPU sharding.

TL2 (DISC 2006). Transactional Locking II [11] introduced commit-time locking with a global version clock. QuasarSTM shards TL2 into per-lane clocks (§3) and uses lane-clock invariance as the Tier-1 fast validation predicate (§4). There is no global clock anywhere in QuasarSTM.

TicToc (SIGMOD 2016). TicToc [19] eliminated the global timestamp counter with per-data-item ($read\_ts, write\_ts$) pairs. QuasarSTM adopts the per-key timestamp pair (§6) to *reduce false aborts*, with the hard rule (Invariant 6.1) that timestamps cannot reorder consensus. Canonical TicToc derives the order from the timestamps; QuasarSTM accepts the order from Quasar and uses timestamps only to justify the absence of an abort.

GPU-STM literature. Cederman et al. [10] and Holey et al. [13] catalogue SIMT livelock and scalability collapse under skewed access as the dominant pathologies of naive GPU-STM ports. QuasarSTM rejects both root causes: deterministic contention management (§9), bounded repair (Lemma 9.2), and Prism refraction (§7) that isolates hot lanes *before* STM sees them.

Aria. Aria [15] executes a batch against a single snapshot and deterministically commits the maximal serializable subset. QuasarSTM 3.0 does *not* adopt this model; the 3.1 evolution (§14.1) introduces an Aria-style commit-selection service.

ESSN / multiversion serialization graphs. Adya et al. [1] formalise validation against a multiversion serialization graph. QuasarSTM 3.0 uses the simpler "visible-version under canonical order" validation; the 3.1 evolution introduces an MVSG-derived rule (`KnownTotalOrder` [16]).

Quasar consensus stack. Quasar [14] is the post-quantum BFT consensus layer that supplies $\preceq_{\text{can}}$ to QuasarSTM. LP-132 [17] is the GPU execution adapter that hosts QuasarSTM (wave-tick scheduler, device-resident rings, EVM fiber VM, async cold-state, per-lane cert verification).

14.1 Adaptive Evolution: 3.1, 3.2, 4.0

QuasarSTM 3.0 (this paper) is the GPU-native ordered MVCC baseline. Ongoing research—out of scope for the 3.0 specification—converges on a hybrid adaptive thesis: predict independence, isolate owned state, defer commutative hot operations, use MVCC only where needed, validate against a known deterministic order, and keep the GPU commit path batched. Three named generations are tracked in the LP-010 living document [16]:

- **3.1**: ConflictSpec ABI (AFT 2025 conflict-specifications work [2]), NEMO [8] LaneClass (owned / shared / hot-shared / commutative), Aria-style deterministic commit selection.
- **3.2**: RapidLane [9] deferred-object reducers, ForeSight [5] predictive scheduler, Chiron [3] execution-hint roots, Multiverse [7] dynamic per-lane versioning mode.
- **4.0**: CSMV [4] GPU client/server commit separation, Motor [6] VersionBlock RDMA layout, vMVCC [18] formally specified CPU reference, surgical operation-level concurrency for pathological long transactions.

These are research, not specification. The 3.0 spec frozen on December 15, 2025 stands on its own.

15 Conclusion

QuasarSTM 3.0 is the third generation of Block-STM and the first to treat the GPU as a primary execution surface rather than a faster CPU. Block-STM 1.0 proved that a known canonical order is a validation primitive, not a serialisation bottleneck. Block-STM 2.0 ported that idea to the GPU. QuasarSTM 3.0 re-architects the entire fabric around the lane abstraction: contention is predicted by hint, prevented by Prism refraction, validated in three tiers (lane-clock, key-MVCC, semantic), repaired incrementally against fiber checkpoints, managed deterministically by a six-policy contention manager with explicit hard limits, and finalised in batches by commit horizons.

The headline result is that Block-STM stops being a repair loop and becomes a serializability fabric. Production targets of $repair\_amplification < 1.01$ on disjoint-lane workloads and controlled degradation under adversarial Zipfian contention are achieved by structure, not by tuning.

The four invariants specified in §12 — determinism, ordered serializability, repair monotonicity, and horizon safety — are the contract this fabric meets with the Quasar consensus engine. The 1024-tx end-to-end stress in ∼150 ms on M1 Max and the textbook 16-same-key contention test (120 conflicts, 120 repairs, 16 commits) demonstrate the contract holds at production scale on commodity hardware.

The QuasarSTM 3.0 specification was frozen on December 15, 2025 and activated on the Quasar 3.0 mainnet on December 25, 2025. The adaptive-evolution path (3.1 through 4.0) is documented in the living LP-010 [16] as ongoing research. The 3.0 specification stands on its own.

References

[1] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. Technical report, MIT Laboratory for Computer Science, 2000. Foundational treatment of MVSG-based serializability; ESSN safe-net later derived in subsequent literature.

[2] AFT 2025 Authors. Conflict specifications for block transactional memory. In *Advances in Financial Technologies (AFT)*, 2025. Declared access information accelerates Block-STM via ConflictSpec ABI; oracle-driven scheduling with Block-STM as fallback.

[3] Chiron Authors. Chiron: Execution hints for catch-up acceleration. Manuscript, 2025. Up to 30% speedup on catch-up paths via execution hint roots; safety does not depend on hints.

[4] CSMV Authors. CSMV: Collaborative multi-version STM on GPUs. In *GPU Computing and Applications*, 2024. GPU client/server commit separation; collaborative on-chip metadata.

[5] ForeSight Authors. ForeSight: Predictive scheduling for deterministic OLTP. Manuscript, 2025. Conflict prediction, multiversion optimisation, and light reordering before execution; avoids blind execute-then-abort.

[6] Motor Authors. Motor: One-RDMA-Round-Trip MVCC on disaggregated memory. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024. VersionBlock layout: consecutive tuples for likely-visible read in one RDMA trip.

[7] Multiverse Authors. Multiverse: Dynamic versioning for mixed-contention workloads. Manuscript, 2025. Always-on multiversioning is expensive; mix versioned and unversioned transactions to keep the common case fast.

[8] NEMO Authors. NEMO: Lane-class execution for high-skew blockchain workloads. Manuscript, 2025. Owned / shared / hot-shared / commutative lane class separation; owned-objects fast path + shared-objects validated path.

[9] RapidLane Authors. RapidLane: Deferred-object execution for contended blockchain workloads. Manuscript, 2025. $\sim$12$\times$ throughput improvement on contended workloads via deferred conflicting computation.

[10] Daniel Cederman, Anders Gidenstam, Phuong Ha, Håkan Sundell, Marina Papatriantafilou, and Philippas Tsigas. Lock-free concurrent data structures and transactional memory for many-core architectures. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, 2010.

[11] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

[12] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2023. Originally arXiv:2203.06871, 2022.

[13] Anup Holey and Antonia Zhai. Lightweight software transactions on GPUs. In *Proceedings of the 43rd International Conference on Parallel Processing (ICPP)*, pages 461–470, 2014.

[14] Zach Kelling. LP-105: Quasar Consensus — chain separation for threshold cryptography and QuasarCert soundness. Technical report, Lux Industries, 2025. Lux research paper, December 2025 spec freeze.

[15] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. In *Proceedings of the VLDB Endowment*, volume 13, pages 2047–2060, 2020.

[16] Lux Core Team. LP-010: Block-STM 3.0 — QuasarSTM. Technical report, Lux Industries, 2025. Living LP. Adaptive-evolution roadmap (3.1, 3.2, 4.0) referenced as research, out of scope for 3.0 spec.

[17] Lux Core Team. LP-132: QuasarGPU execution adapter. Technical report, Lux Industries, 2025. GPU-native execution adapter for Quasar-certified rounds: wave-tick scheduler, device-resident rings, EVM fibers, MVCC Block-STM, async cold-state, per-lane cert verification.

[18] vMVCC Authors. vMVCC: Machine-checked MVCC linearisation. Manuscript, 2024. Formally verified MVCC reference; subtle linearisation corner cases.

[19] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 1629–1642, 2016.

Reproducibility

Source code. `github.com/luxfi/consensus` (commit `TBD`); GPU kernels at `github.com/luxfi/gpu`; STM runtime at `github.com/luxfi/stm`.
Build. `go build ./...`; GPU paths require CUDA $\geq$ 12 or Metal toolchain on macOS.
Hardware tested. TBD (multi-GPU x86_64 Linux; Apple Silicon for the Metal back-end).
Standards referenced. FIPS 204 (ML-DSA) for the PQ lane; project-internal LP-010, LP-020, LP-070.
Contact. security@lux.network.