

Lemonshark: Asynchronous DAG-BFT With Early Finality

Michael Yiqing Hu
National University of Singapore
hmichael@nus.edu.sg

Yihan Yang
National University of Singapore
yangyihan@comp.nus.edu.sg

Alvin Hong Yao Yan
National University of Singapore
dcsayhy@nus.edu.sg

Xiang Liu
National University of Singapore
liu.xiang@u.nus.edu

Jialin Li
National University of Singapore
lijl@comp.nus.edu.sg

Abstract

DAG-Rider popularized a new paradigm of DAG-BFT protocols, separating dissemination from consensus: all nodes disseminate transactions as blocks that reference previously known blocks, while consensus is reached by electing certain blocks as leaders. This design yields high throughput but confers optimal latency only to leader blocks; non-leader blocks cannot be committed independently.

We present Lemonshark, an asynchronous DAG-BFT protocol that reinterprets the DAG at a transactional level and identifies conditions where commitment is sufficient—but not necessary—for safe results. Enabling nodes to finalize transactions before official commitment without compromised correctness. Compared to Bullshark, the state-of-the-art asynchronous BFT, Lemonshark reduces latency by up to 65%.

1 Introduction

Modern distributed applications require consensus protocols that can achieve both high throughput and low latency in adversarial environments. From cryptocurrency networks [1, 5, 6] processing millions of transactions daily to geo-distributed databases [36, 40, 42, 43] serving global user bases, these systems must maintain consistency despite potential node failures and malicious behavior.

Byzantine Fault-Tolerant (BFT) consensus protocols [17, 18, 23] address this challenge by solving the Byzantine Atomic Broadcast (BAB) problem [21]: ensuring all honest processes agree on a total order of messages (e.g., transactions) despite Byzantine faults, with both safety and liveness guarantees.

While BFT protocols solve this fundamental problem, classical Asynchronous BFT approaches [24, 35, 37, 39] utilize paradigms [14, 15, 18] that have been perceived as either excessively complex or prohibitively costly for practical deployment. While recent advances have substantially improved

performance [10, 37], a novel class of DAG-based protocols [13, 26] has emerged, introducing an innovative paradigm that promises optimal round and amortized communication complexity [22, 27, 31].

DAG-based BFT protocols [22, 27, 31] follow the principle of decoupled data dissemination and consensus to improve throughput and expected latency. These protocols enable every node to propose transactions in the form of blocks that reference previously known blocks, collectively forming a Directed Acyclic Graph (DAG) across rounds. This parallelization is the key to their increased throughput. While the DAG imposes only a partial order, a total order can be derived by selecting specific blocks as leaders.

DAG-BFT protocols achieve consensus by electing leader blocks using a Global Perfect Coin abstraction [22, 31]. This mechanism significantly improves expected latencies compared to traditional approaches [24, 35, 37, 39]. Once confirmed, leaders and their causal histories (all blocks reachable through directed paths) are committed, then ordered deterministically and executed to produce **finalized** outcomes for the constituent transactions of blocks.

Prevailing designs couple *finalized* outcomes with block *commitment*. The leader election process fundamentally governs block commitment. *Elected leaders may achieve optimal latency* (committed after the next round), while non-leader blocks require inclusion in the causal history of a subsequently elected leader to be committed. This asymmetry creates a latency disparity: even under perfect network conditions without faults, non-leader blocks experience 1-2 additional rounds of delay compared to leader blocks.

Moreover, leader elections typically yield a single leader and occur infrequently, thereby exacerbating latencies for the majority of blocks. The current state-of-the-art asynchronous DAG-BFT protocol, Bullshark [27], has only partially ad-

addressed this issue by enabling more frequent leader elections during periods of relative progress.

Rather than accelerating leader election, Lemonshark introduces a theoretical shift:

*We may yield **finalized** outcomes for transactions in a non-leader block **before** it is committed.*

We term this capability **Early Finality** — which decouples finalization from commitment. Lemonshark demonstrates that commitment via inclusion in a committed leader block’s causal history represents a sufficient, but not necessary, condition for obtaining finalized outcomes when it comes to transactions in asynchronous DAG-BFT protocols.

Naively attempting to derive *finalized* outcomes for transactions contained within non-leader blocks before their *commitment* is generally unsafe. In asynchronous, adversarial environments, a node’s local view of the DAG may not be complete. As such, there may exist blocks unknown to the node—whose transactions conflict with or otherwise influence outcomes of those in its local view. Even if the node’s local view includes all the blocks, the inclusion of a given block in the causal history of a subsequently elected leader cannot always be guaranteed.

Lemonshark addresses these safety challenges through a key insight: knowledge of the ordering of conflicting blocks within the eventual committed leader’s history enables the determination of safe, finalized outcomes even before the leader block is proposed. We implement this through key-space partitioning and deterministic causal history ordering. Lemonshark partitions the key-space across blocks within each round, ensuring that each block operates on a distinct shard and eliminating intra-round conflicts. This constraint, combined with predefined ordering rules, enables nodes to safely predict final transaction outcomes by observing only conflicting actions from previous rounds — a practically achievable requirement even in network asynchrony and with Byzantine adversaries. However, partitioning key spaces can lead to transactions that span multiple shards. While key-space sharding introduces complexity for cross-shard transactions, Lemonshark can achieve early finality by re-interpreting the DAG locally, without any additional coordination. Failing the checks only delays finalizing transactions to the original commitment time, which has no additional throughput or latency penalties.

To the best of our knowledge, Lemonshark is the only asynchronous DAG-BFT protocol that provides early finality. We empirically compared Lemonshark with Bullshark, the state-of-the-art asynchronous BFT protocol, in a geo-distributed setting across five AWS regions. Lemonshark attains up to 65% lower consensus latency in the failure-free case, while achieving virtually equivalent maximum throughput. Even under adversarial conditions, Lemonshark maintains its early finality benefits. In the presence of a single node failure, Lemonshark attains an approximate 50% improvement; even under maximum tolerable failures, Lemonshark still main-

tains an approximate 24% latency benefit.

2 Preliminaries

We consider a set of n nodes: $\Pi = \{p_1, \dots, p_n\}$, where a dynamic computationally-bounded adversary can corrupt up to $f < \frac{n}{3}$ unique nodes. Corrupted nodes are considered *Byzantine*, whilst the remainder are regarded as *honest*. Byzantine nodes may exhibit arbitrary behavior, whereas honest nodes follow the protocol faithfully. We assume a public key infrastructure exists for node identity verification.

We adopt identical timing assumptions as Bullshark: we make no timing assumptions, and progress occurs even under full network asynchrony. Messages may be reordered or delayed arbitrarily, but are eventually delivered.

Since Lemonshark utilizes identical dissemination and consensus mechanisms as Bullshark, we assume a reliable broadcast (RBC) [17] primitive with agreement, integrity, and validity properties [27]. We also assume a Global Perfect Coin [16, 34, 44] primitive for randomized leader election to circumvent the FLP impossibility [25].

3 Background and challenges

DAG-based BFT protocols represent the current state-of-the-art in asynchronous BFT, with Bullshark being the most performant implementation. Since Lemonshark utilizes the same state-of-the-art core consensus mechanism, we first examine its design before analyzing the fundamental challenges that prevent early finality.

3.1 DAG Core

DAG-BFT protocols, such as Bullshark [27], decouple message dissemination from consensus. The dissemination protocol operates in sequential rounds $r = 1, 2, 3, \dots$, where each node performs reliable broadcast (RBC) [17] of a message containing:

- A unique node identifier.
- A set of client-submitted transactions.
- **Pointers** to $(\geq 2f + 1)$ blocks from the previous round.
- Some additional metadata.

Transactions represent atomic operations that read and modify key-value pairs in a shared state. Upon successful RBC completion, the message becomes established as a block b . **The RBC primitive provides two crucial guarantees:** eventual availability of the block to all correct nodes, and identical block delivery across correct nodes (non-equivocation). These blocks serve as vertices in a graph structure, with their pointers to previous-round blocks forming edges, collectively constructing the DAG. In effect, this strong primitive mitigates the most disruptive aspects of Byzantine behavior, ensuring that malicious nodes can only interfere with the protocol by remaining silent, much like a crashed node. However, it is possible for a block to become *orphaned* if no subsequent block in the following round points to it. We elaborate in Appendix D how Lemonshark handles such blocks.

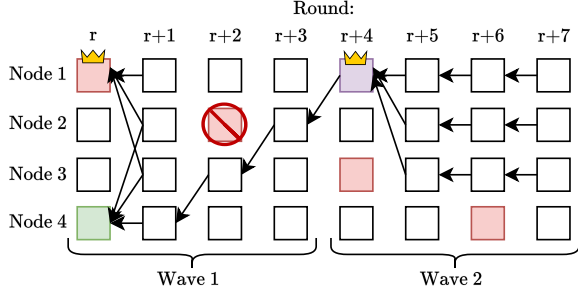


Figure 1: An example of latency disparities in Bullshark. Only critical pointers are shown for clarity. Red blocks denote steady leaders, purple blocks denote fallback leaders, with crowns indicating committed leaders. The steady leader at round $r+1$ commits upon obtaining sufficient votes, while the green block at round r must await inclusion in a future leader’s causal history, incurring 8 additional rounds of delay. Were the green block itself a steady leader, it would commit at round $r+1$.

3.1.1 Consensus Core

Under network asynchrony, nodes may maintain divergent local views of the global DAG. However, since reliable broadcast precludes equivocation, Byzantine atomic broadcast is achieved by reaching consensus on a totally ordered list of blocks *designated as leaders*.

The challenge lies in achieving agreement on which blocks are selected as leaders. Bullshark specifically employs two leader types, where at most one type may commit within each 4-round *wave*:

- **Steady leaders:** Selected deterministically via round-robin every 2 rounds.
- **Fallback leaders:** Selected randomly using a global perfect coin (instantiated with threshold signatures [16, 34, 44]) every 4 rounds to circumvent the FLP impossibility [25].

When referring to a leader, we mean a block with that allocated designation. Lemonshark utilizes the same consensus mechanism as Bullshark. The leader selection process works as follows: in each wave, blocks produced by a particular node and its pointers constitute either a vote for a *steady* or *fallback* leader depending on whether progress was witnessed as part of its causal history in the previous wave. A steady leader *commits* when at least $2f+1$ blocks from the subsequent round have pointers to it. Fallback leaders are identified and committed through the voting process at wave completion. Optimal latency is enjoyed by a steady leader that obtains sufficient votes in the immediately following round.

This leader commitment process is illustrated in Fig. 1, where the blocks result from reliable broadcast, the red blocks are steady leaders, and the purple blocks are fallback leaders. The steady leader in $r+1$ obtained sufficient votes and commits, while the missing steady leader in $r+3$ impedes the progress, leading to all block paths in the subsequent wave to be the fallback votes.

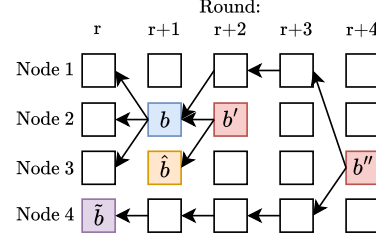


Figure 2: Both \hat{b} and \tilde{b} contain transactions conflicting with those in b . This figure demonstrates that Bullshark requires commitment to resolve ordering uncertainty for such blocks.

3.1.2 Ordering and Executing Transactions

The consensus core (§3.1.1) produces a totally ordered list of leaders. We define a block’s *causal history* as the set of blocks it has paths to, excluding those already committed by previous leaders. A node’s view of any block’s causal history may vary depending on its perception of the last committed leader. Each leader block’s causal history is sorted using a deterministic algorithm and ordered in the same sequence as the initial totally ordered list.

When a new leader b' commits, its sorted causal history is appended to this sequence. Transactions within each block are then executed in this order, yielding *finalized* outcomes for the corresponding transactions and blocks. Prior works [22, 31] employ different leader commitment mechanisms with varying election frequencies. However, the resulting leader list and transaction execution semantics remain analogous.

3.2 Inequitable Finality

While the ordering and execution mechanism described above ensures consistency, it creates significant latency disparities between different types of blocks. For a block to commit, it must either be part of a committed leader’s causal history or be a leader itself. Since only a single leader may be elected every 2 rounds, most blocks experience longer commitment latencies than leaders. Even under optimal network conditions, non-leader blocks experience 1-2 additional rounds of delay compared to leaders, as they must await inclusion in a future leader’s causal history.

This creates an inherent asymmetry where transaction latency depends not on network conditions or system load, but solely on the arbitrary assignment of transactions to leader versus non-leader blocks.

3.3 Challenges in Achieving Early Finality

This asymmetry raises a fundamental question: can we determine a transaction’s finalized outcome before its containing block commits—circumventing the leader election bottleneck? This section demonstrates why early finality determination is unsafe in Bullshark, with formal proofs in Lemma A.1.

The core challenge stems from network asynchrony and Byzantine adversaries, which create uncertainty about the complete set of conflicting transactions that may execute before a given block. We illustrate this using Fig. 2: Consider

Node 2 at round $r + 1$ producing block b . Its local DAG view includes b 's causal history and at least $2f$ blocks from other nodes in round $r + 1$. Due to asynchrony and potentially inactive Byzantine nodes, Node 2 proceeds to round $r + 2$ without awaiting all n blocks from round $r + 1$.

This introduces ordering uncertainty: a block \hat{b} outside Node 2's view may contain transactions affecting b 's outcome if \hat{b} precedes b in execution order. Since conventional DAG-BFT protocols impose no restrictions on transaction-to-block assignment, Byzantine adversaries can introduce such blocks in any round, modifying every system key. If \hat{b} obtains insufficient pointers ($< f + 1$) in round $r + 2$, we cannot guarantee that a future leader b' (that has a path to b) at round r' will have a path to \hat{b} . Thus, we must first witness b' .

However, even this is insufficient. After observing b' at round r' , uncertainty remains about whether b' will commit; b' may fail to secure sufficient votes, and a different eventual leader (b'') may have a different view regarding \hat{b} 's effect on b —potentially excluding \hat{b} from its causal history entirely.

These compounding uncertainties yield our main result: without fundamental modifications to how the DAG structure is perceived, early finality is unachievable under network asynchrony with Byzantine nodes.

4 Problem Definition

The ineluctable finality described in §3.2 illustrates how arbitrary transaction-to-block assignments lead to the uneven latencies when obtaining finalized outcomes. This motivates the central question of Lemonshark: **Can we determine a transaction's outcome before it is committed?** Thereby providing lower latencies for all blocks! This section formalizes the notion of *Early Finality* and establishes the theoretical framework for Lemonshark.

4.1 Refined Definition of Causal History

We begin by providing a concrete definition of causal history, specifying the exact ordering mechanism:

Definition 4.1 ((Sorted) Causal History of a Block). *For a block b , its causal history is the set of nodes that are part of a sub-DAG, where b is the root, exclusive of the blocks that are in the previous known committed leader's causal history. Its sorted causal history is obtained by applying Kahn's algorithm [30] to the sub-DAG and reversing the list, where ties (blocks of the same round) are broken deterministically. We denote this list as $H_b = [\dots, b]$.*

Unlike existing DAG-BFT protocols that accept any deterministic ordering method, we impose a temporal constraint on the ordering: blocks from earlier rounds must be ordered before those from later rounds in a block's causal history (Fig. 3 illustrates this concept). This ordering choice, while intuitive, is **critical** to Lemonshark's early finality guarantees—arbitrary orderings would undermine our ability to predict transaction outcomes before commitment.

When a leader b' commits, all blocks in $H_{b'}$ are committed and executed sequentially. Non-leader blocks are committed

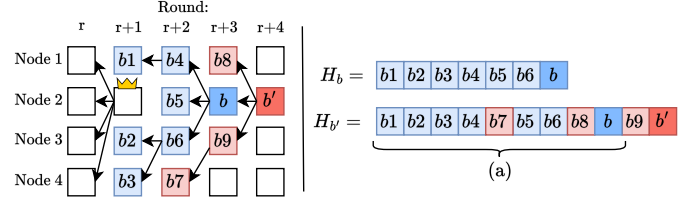


Figure 3: Blue blocks represent the causal history of b —excluding previously committed blocks. Blocks are ordered via breadth-first traversal by ascending round number, terminating at b . (a) Executing blocks in this order (b_1, \dots, b) yields the execution prefix $b'\langle b \rangle$.

if and only if they appear in some committed leader's causal history. For a committed leader b' , we say it *commits* b for all $b \in H_{b'}$ as well as all transactions in b . Given the strict monotonic ordering between committed leaders, for leaders b' and b'' that are committed consecutively, all elements in $H_{b'}$ are committed before those in $H_{b''}$. It is also important to note that a block b can only be *committed* by a single leader. Once committed by a leader, b is excluded from the causal history of any other leader.

4.2 Transaction and Block Outcomes

To evaluate early finality, we first define outcomes for transactions and blocks:

Definition 4.2 (Transaction Outcome (TO)). *For a block $b = [t_1, \dots, t_i, \dots, t_p]$, the transaction outcome (TO) of t_i is the outcome of t_i when executing transactions in the following order: $H_b[-1] + [t_1, \dots, t_i]$. We denote the prefix of H_b exclusive of b as $H_b[-1]$.*

Definition 4.3 (Block Outcome (BO)). *For a block b and its Sorted Causal History H_b , the Block Outcome (BO) of b is the execution results of all transactions in b after executing the blocks in the order of H_b .*

The rationale for evaluating transaction and block outcomes based on its own causal history is to eliminate intra-round dependencies.

4.3 Early Finality

To this end, we define *Execution Prefix* as the outcome of intermediary blocks within a block's sorted causal history $H_{b'}$, computed prior to deriving the BO of b' . This serves as our reference point for comparison.

Definition 4.4 (Execution Prefix (Block)). *For blocks $b', b : b = [t_1, \dots, t_i, \dots, t_p]$, where $b \in H_{b'}$. The execution prefix $b'\langle b \rangle$ is the outcome of the transactions in b when executing the prefix of $H_{b'}$ up to and including b (i.e., $H_{b'}[0 : \text{index}(b)]$ or $H_{b'}[0 : \text{index}(b) - 1] + [t_1, \dots, t_i, \dots, t_p]$). We describe this as the execution prefix of b with respect to b' .*

Definition 4.5 (Execution Prefix (Transactions)). *For blocks $b', b : b = [t_1, \dots, t_i, \dots, t_p]$, where $b \in H_{b'}$. The execution prefix $b'\langle b(t_i) \rangle$ is outcome of transaction t_i when executing the prefix of $H_{b'}$ right before b and $[t_1, \dots, t_i]$ (i.e., $H_{b'}[0 : \text{index}(b) - 1] + [t_1, \dots, t_i]$). We describe this as the execution prefix of t_i with respect to b' .*

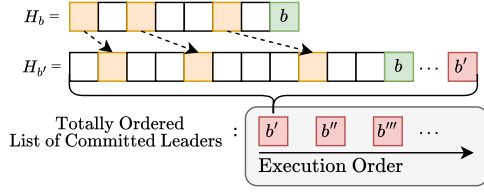


Figure 4: Orange blocks represent all uncommitted blocks from rounds prior to b 's round that might impact the *finalized* outcome of b . The totally ordered list of committed leaders is generated by the DAG consensus core (§3.1).

When the leader block b' (where $b \in H_{b'}$) obtains sufficient votes, the execution prefix of each block/transaction in $H_{b'}$ with respect to b' **becomes the *finalized* immutable outcome**.

Suppose a non-leader block's BO matches the execution prefix with respect to the leader that commits it. In that case, deriving the BO is functionally equivalent to deriving the block's eventual finalized outcome. We formalize this as Safe Transaction/Block Outcome:

Definition 4.6 (Safe Transaction Outcome (STO)). *For a particular transaction $t_i \in b$, we say that its transaction outcome is safe if for a committed leader block $b' : b \in H_{b'}$, the transaction outcome of t_i is equivalent to the execution prefix $b' \langle b(t_i) \rangle$.*

Definition 4.7 (Safe Block Outcome (SBO)). *If all transactions within a block b have STO, we say the block has a safe block outcome (SBO).*

Finally, we define our central objective of Lemonshark:

Definition 4.8 (Early Finality). *For a non-leader block b , early finality is achieved for b if the SBO of b may be determined before b is determined to be committed.*

5 Lemonshark

Lemonshark builds upon the block dissemination and consensus mechanisms described in §3.1, but restructures block content and reinterprets the DAG structure to enable early finality evaluation.

As demonstrated in §3.3, early finality was unattainable because blocks potentially ordered before b —which may contain conflicting transactions—remain ambiguous until the leader committing b is determined.

Lemonshark addresses this fundamental limitation through a key insight (illustrated with Fig. 4): if a block b 's causal history includes all uncommitted blocks from prior rounds that can affect its execution outcome, when b is committed by the leader b' , our ordering constraint (Definition 4.1) ensures that only blocks from the same round as b may affect b 's execution prefix with respect to b' . If those blocks indeed *do not* affect b 's execution, SBO for b can be evaluated before it is known to be committed.

In Lemonshark, this determination is made independently by each node using its local view of the DAG. It is important to note that Lemonshark does not enforce the conditions

for early finality; rather, each node surveys its local DAG and identifies blocks (such as b) that have met the criterion for SBO. All honest nodes will eventually make the same determination or, once the corresponding leader block b' is committed, derive identical *finalized* outcomes for all transactions in b . When SBO for b is determined before commitment of b' , the block can be executed early, delivering finalized results to clients and reducing perceived latency.

In this section, we present the sufficient conditions that nodes can check locally to determine whether early finality can be achieved for a block, enabling safe execution results before commitment.

5.1 Sharded Key-Space

As discussed in §3.3, each node must proceed after witnessing $\geq 2f + 1$ blocks in round r , since waiting indefinitely for potentially inactive Byzantine nodes is impractical.

Consequently, when a node produces block b in round $r + 1$, it cannot assert that all blocks prior to round $r + 1$ containing transactions affecting b 's execution prefix with respect to its eventual committing leader are included in b 's causal history.

Lemonshark addresses this limitation by requiring each node to operate on a distinct, rotating partition of the key-space within each round. Specifically, we partition the key-space over which transactions operate into n disjoint shards: $K = \{k_1, \dots, k_n\}$.

In each round, only a single node may produce a block containing transactions that modify keys of a particular shard. Furthermore, the node-to-shard mapping rotates each round according to a publicly known schedule. For example, node p_i in-charge of shard k_i at round r becomes in-charge of shard $k_{(i+1) \bmod n}$ at round $r + 1$. A block containing exclusively transactions that write to keys in shard k_i is designated as being **in-charge** of that shard. For notational clarity, we denote b_i^r as a block from round r that is in-charge of shard k_i . Clients may broadcast their transactions to all nodes, ensuring that a node in-charge of that key-space will be able to handle it immediately upon receiving it.

This rotation scheme prevents censorship attacks and simplifies dependency tracking: block b only needs to consider uncommitted blocks from previous rounds that operated on the same shard, rather than all potentially conflicting blocks.

We assume the key-space partitioning scheme achieves load balance while minimizing cross-shard transactions [29, 41]—The specific mechanisms for achieving optimal partitioning are beyond the scope of Lemonshark.

Key-space partitioning introduces cross-shard transaction complexity. To demonstrate Lemonshark's generality, we define three transaction types that cover the common cross-shard access patterns:

- **Type α :** Intra-shard transactions that read and write exclusively within shard k_i
- **Type β :** Cross-shard read transactions that read from multiple shards but write only to shard k_i

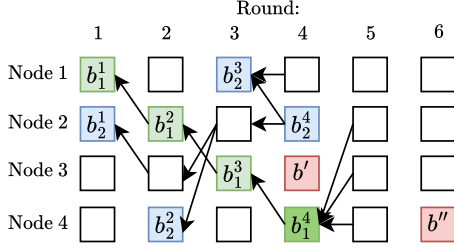


Figure 5: Consider a scenario where all blocks before round 1 has committed. The blue and green blocks are in-charge of two distinct shards (k_1, k_2), and the red blocks b', b'' are leader blocks. Suppose both b_1^4, b_2^4 persists in round 5. For b_2^4 , it has all uncommitted blue blocks as part of its causal history; however, each block does not point to the block of the same shard in the previous round. The block outcome of b_2^4 will execute blue blocks in the order of: $(b_2^1, b_2^2, b_2^3, b_2^4)$. However, if $b_2^2 \in H_{b'}$, $b_2^1, b_2^3 \notin H_{b'}$ and b' is committed, then the final order is $b_2^2, b_2^1, b_2^3, b_2^4$; therefore, b_2^4 does not have SBO. In contrast, suppose b_1^3 has SBO, and b_1^4 points to it and persists in round 5, its BO will execute the green blocks in the order: $(b_1^1, b_1^2, b_1^3, b_1^4)$. Regardless of which subset of green blocks is in $H_{b'}$, the block outcome of b_1^4 will always be equivalent to the execution prefix $b'' \langle b_1 \rangle$.

- **Type γ :** Atomic multi-shard transactions consisting of coordinated Type α/β sub-transactions that maintain serializability.

These transaction types cover essential database operations [20, 33, 47]: local updates (α), cross-shard reads (β), and atomic coordination (γ). Extensions to additional transaction types are possible, but beyond our current scope.

For clarity, we focus on basic forms: Type β transactions that read from a single other shard, and Type γ transactions that are sub-transaction pairs. Extensions to Type β transactions reading from arbitrary numbers of shards and Type γ transactions as n -tuples are detailed in Appendix B.

The remainder of this section introduces each transaction type before examining the intricacies that Lemonshark must address to enable their early finality.

5.2 Type α : Intra-Shard Transactions

Type α transactions represent the optimal case for our Lemonshark, performing both reads and writes exclusively within a single shard. This subsection demonstrates how blocks containing only Type α transactions can achieve SBO before commitment, building the foundation for the more complex transaction types.

We proceed inductively, first analyzing how the earliest uncommitted block in-charge of a shard achieves SBO. We then extend this analysis to subsequent blocks, carefully examining necessary conditions and potential edge cases that arise from evaluating early finality.

5.2.1 First Uncommitted Block

To inductively establish whether a block in-charge of a shard achieves SBO, we begin with the first uncommitted block in-charge of that shard, illustrated by block b_1^1 in Fig. 5.

This block possesses a unique advantage: if no leaders

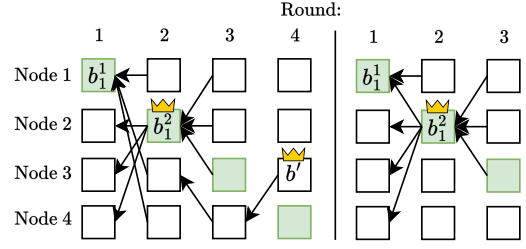


Figure 6: Suppose green blocks are in-charge of a unique shard (k_1). On the left portion of the figure, b' eventually commits b_1^1 . This figure illustrates two ways the restriction discussed in §5.2.2 may be bypassed.

exist in the subsequent round and it is committed by a leader b' from round 3 or later, it can guarantee its position as the first block in-charge of shard k_1 within the ordering of $H_{b'}$. Our ordering constraint (Definition 4.1) ensures that no other block in-charge of shard k_i may precede b_1^1 in $H_{b'}$.

However, b_1^1 must still ensure its eventual inclusion in a leader's causal history. This requirement is satisfied by achieving *persistence*. A block *persistence* at round r if $> f + 1$ blocks from that round have paths to it. Since each block contains $\geq 2f + 1$ pointers to blocks from the previous round, quorum intersection ensures that if a block persists at round r , all blocks from round $r + 1$ onwards must have a path¹ to it.

Consequently, if b_1^1 persists at round 2, any leader from round 3 onwards is guaranteed to have a path to it. Therefore, absent a leader at round 2 and given that b_1^1 persists at round 2, its BO equals the execution prefix with respect to any leader from round ≥ 3 that eventually commits it. Since the earliest such leader commits at round 4, exceeding the round at which b_1^1 persists (round 2), early finality is achievable for the first uncommitted block in-charge of a shard containing exclusively Type α transactions.

5.2.2 Leader in the Next Round

In §5.2.1, early finality was achieved for b_1^1 under the assumption that no leader exists in round 2. However, suppose b_2^2 is a leader that commits without having a path to b_1^1 (illustrated in the left portion of Fig. 6), where as the latter is eventually committed by b' . In this case, we cannot assert that the BO of b_1^1 is safe, as execution follows the totally ordered list of committed leaders, possibly placing b_2^2 before b_1^1 in execution order. We now discuss sufficient conditions to circumvent this situation.

One approach is to wait until leader b_1^2 commits: Once b_1^2 is known to be committed, we can guarantee that no other block in-charge of k_1 will precede b_1^1 in $H_{b'}$. We formally show this in Proposition A.5. Since the earliest b_1^2 may commit is at round 3, and the next leader (b') can only exist from round 4 onwards, early finality remains achievable for block b_1^1 .

Alternatively, if b_1^1 has a pointer to b_1^1 (illustrated in the right portion of Fig. 6), then committing b_2^2 also commits b_1^1 . Since b_1^1 is from an earlier round, our ordering method

¹ This requirement aligns with the leader-election criterion for the partially synchronous version of Bullshark [46].

guarantees it executes before b_1^2 . This generalizes as follows: for a block b_i^r , if b_i^{r+1} is a leader with a pointer to b_i^r , then b_i^r executes before b_i^{r+1} when committed.

How can we determine which blocks may be elected as leaders in round $r+1$? At each wave's beginning, the labeling of steady and fallback votes across rounds is predetermined, enabling identification of potential committed steady leaders per wave. However, determining potential fallback leaders is not possible *a priori*. Consequently, when a fallback leader could potentially commit, we conservatively assume any block in the wave's first round could become a committed fallback leader. In this case, we require b_i^{r+1} to have a pointer to b_i^r , guaranteeing that b_i^{r+1} will not execute before b_i^r . We formalize this requirement in Proposition A.4.

We denote all conditions specified in this subsection as *leader-checks* for shard k_i . Specifically, if either condition is satisfied for a block b from round r with respect to block b_i^{r+1} , we say that it passes the leader-checks for shard k_i . We summarize and illustrate the detailed conditions for leader-checks in Algorithm A-1.

5.2.3 Subsequent Uncommitted Blocks

We now demonstrate how blocks other than the earliest uncommitted ones in-charge of a shard may achieve early finality. Reusing the conditions specified in Sections 5.2.1 and 5.2.2 is insufficient. We illustrate this limitation by examining b_2^3 in Fig. 5. Suppose b_2^3 persists in round 4, passes the leader-check for shard k_2 and is eventually committed by b'' . Further suppose b_2^2 does not persist in rounds 3, making it impossible to determine whether b_2^2 will be in $H_{b''}$ preceding b_2^3 in execution. This is because although b_2^3 ensures its eventual commitment by persisting in round 4, the fate of uncommitted blocks in-charge of k_2 prior to round 3 remains uncertain.

A straw-man solution would require all uncommitted blocks prior to round 3 and in-charge of k_2 to be part of b_2^3 's causal history, but we illustrate in Fig. 5 that this may not be sufficient. Similar to the problem described in §3.3, having complete knowledge of all blocks locally does not necessarily imply the execution order. A leader's causal history might include only a subset of those blocks, resulting in a completely different execution order from the one in b_2^3 's BO.

However, if we can enforce that a committed leader *must* include a *proper prefix* of those blocks, then we can ensure the eventual execution is consistent with b_2^3 's BO. Lemonsark addresses this challenge by requiring block b_i^r to have a pointer to b_i^{r-1} , where the latter has SBO. This creates a recursive path of blocks from b_i^r to the oldest uncommitted block in-charge of k_i in round \hat{r} , traversing blocks from rounds $\hat{r}+1$ to $r-1$ that are in-charge of k_i . When such a path exists, we demonstrate in Fig. 5 that this structure is sufficient to ensure that any committed leader must include a proper prefix of this chain; it may not contain an arbitrary subset of blocks from this chain within its causal history.

More generally, if b_i^r points to b_i^{r-1} where the latter has SBO, persists in $r+1$, and passes the leader-check for k_i , we

Algorithm 1 α -STO Eligibility Check

```

1: function  $\alpha$ -STOCHECK( $t \in b_i^r$ )
2:   if  $\exists t' \in DL_r$  where  $t$  modifies same key as  $t'$  then
3:     return false
4:   end if
5:   if not LEADERCHECK( $b_i^r, k_i$ ) then
6:     return false
7:   end if
8:   return ( $(b_i^r$  is oldest uncommitted block in-charge of  $k_i$ ) or ( $b_i^r$  points
      to  $b_i^{r-1}$  and  $b_i^{r-1}$  has SBO)) and ( $b_i^r$  persists in  $r+1$ )
9: end function

```

Note: DL_r will be elaborated further when we discuss Type γ transactions in §5.4.3. LEADERCHECK() is described with §5.2.2 and illustrated in Algorithm A-1.

can guarantee that no block in-charge of k_i that is not contained in b_i^r 's causal history can execute before b_i^r . Therefore, we can ensure that the BO of b_i^r will match the execution prefix with respect to the leader that eventually commits it and thus have SBO.

Since the earliest for a non-leader block b_i^r to be committed is in round $r+2$, while learning that it persists can be achieved in $r+1$, early finality remains possible. We illustrate all sufficient conditions for a Type α transaction to have STO in Algorithm 1 and provide formal proofs in Lemma A.2. We also demonstrate in Proposition A.3 that since a substantial subset of at least $\frac{3f+2}{2}$ blocks from each round *must* persist in the subsequent round, early finality for some blocks remains achievable even in the presence of byzantine adversaries and network asynchrony.

5.3 Type β : Cross-shard Read Transactions

Type β transactions differ from Type α transactions by reading from a different shard that they write to. Since Type β transactions continue to write to the shard that their containing block is in-charge of, they retain the same requirements as Type α transactions. Additionally, we must ensure that the perceived read value when evaluating TO matches that obtained when evaluating the execution prefix of the transaction with respect to the leader that eventually commits it.

In this subsection, we specifically examine how a particular block from round r may exhibit characteristics that accommodate possible modifications to the read value occurring before, during, and after r . We then identify conditions sufficient for a Type β transaction within it to achieve early finality. To aid in our illustration, consider a Type β transaction t in a non-leader block b_i^r . It reads from k_j : $k_j \neq k_i$ and is eventually committed by a leader block b' .

5.3.1 Read Value Before r

Similar to §5.2.3, we must ensure that all possible uncommitted blocks modifying k_j prior to round r execute before b_i^r . This is achieved by having b_i^r point to b_j^{r-1} , where the latter has SBO. We can then assert: no uncommitted blocks in-charge of k_j existing before round r will execute before b_i^r . Alternatively, if the oldest uncommitted block in-charge of k_j is at round $\geq r$, then no block in-charge of k_j prior to round r

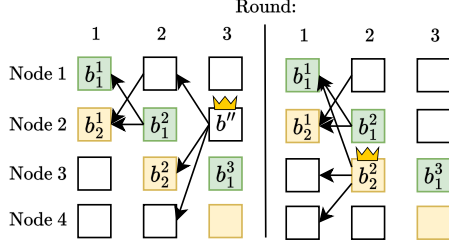


Figure 7: Suppose the green and yellow blocks are in-charge of two distinct shards, k_1, k_2 respectively, no blocks exist before round 1, and leader b' eventually commits b_1^2 . This figure illustrates the two cases in which b_2^2 may be committed while b_1^1 is not—a case for Type β transactions.

may be ordered before b_i^r in execution.

5.3.2 Read Value During r

Our ordering procedure (Definition 4.1) orders blocks of the same round in a mere deterministic order. Therefore, it is possible that b_j^r exists in $H_{b'}$ and is ordered before b_i^r . We do not make any ordering assumptions for blocks within the same round. Yet, we have to ensure that b_j^r does not affect the execution prefix of t with respect to b' .

Instead, if a node were to witness b_j^r in its local DAG, it has to verify if it contains any transactions that write to the key that t reads. If it does not, then it need not worry if any transaction in b_j^r were to affect the execution prefix of t with respect to b' . However, if it does, b_j^r must be committed by a different leader b'' in a round earlier than that of b' , so as not to affect the execution prefix of t with respect to b' . This condition is analogous to the first one described in §5.2.2. There exist some intricacies in **when** exactly b_j^r is committed. We illustrate 2 cases in Fig. 7, focusing on the green block b_1^2 . Suppose b_1^2 has already satisfied all requirements described in §5.3.1 and contains a Type β transaction (t) that reads from the yellow shard k_2 .

In the first case (depicted in the left portion of Fig. 7), $b_2^2 \in H_{b''}$, where b'' is a committed leader. Furthermore, b_2^2 does not have a path to b_2^1 and not b_1^2 , but b_1^1 has a path to b_2^1 (where the latter has SBO). Per §5.3.1, b'' must therefore have a path to b_2^1 . At this point, all blocks before round 3 that are in-charge of k_2 have been committed, and we can be assured that no block in-charge of k_2 before round 3 will supersede b_1^1 in $H_{b'}$. In the second case (depicted in the right portion of Fig. 7), b_2^2 is a committed leader. Regardless of whether b_2^2 has a path to b_2^1 , once b_2^2 is known to be committed, b_2^1 satisfies the second condition of leader-checks on k_2 (per §5.2.2) and t achieves STO. Similarly to the previous case, we can be assured that no block in-charge of k_2 before round 3 will supersede b_1^1 in $H_{b'}$ that does not exist in $H_{b_2^2}$.

5.3.3 Read Value After r

Lastly, similar to §5.2.2, we must consider the case where block b_j^{r+1} may contain transactions that modify the key t reads and is committed before b'/b_i^r . In such a case, we simply perform the same leader-checks for b_i^r but on shard k_j . These

Algorithm 2 β -STO Eligibility Check

```

1: function  $\beta$ -STOCHECK( $t \in b_i^r, k_j : k_j \neq k_i$ )
2:   if  $\exists t' \in DL_r$  where  $t$  modifies same key as  $t'$  or not  $\alpha$ -STOCHECK( $t \in b_i^r$ ) then
3:     return false
4:   end if
5:   if not (( $b_j^r$  is oldest uncommitted block in-charge of  $k_i$ ) or ( $b_i^r$  points to  $b_j^{r-1}$  and  $b_j^{r-1}$  has SBO)) then
6:     return false
7:   end if
8:   if  $b_j^r$  modifies key that  $t$  reads and  $b_j^r$  not yet committed then
9:     return false
10:  end if
11:  return LEADERCHECK( $b_i^r, k_j$ ) or not  $\exists t' \in b_j^{r+1}$  that modifies what  $t$  reads
12: end function

```

Note: DL_r will be elaborated further when we discuss Type γ transactions in §5.4.3. LEADERCHECK() is described in §5.2.2 and illustrated with Algorithm A.1.

checks can be omitted if b_j^{r+1} does not contain transactions that modify the key t is reading, since even if it precedes b_i^r in execution, it does not affect the execution prefix of t with respect to b' .

Collectively, with the conditions mentioned in Sections 5.3.1 and 5.3.2, we can ensure that blocks in-charge of k_j from rounds before, during, and after r will not impact the execution prefix of t with respect to b' . We illustrate the sufficient conditions for a Type β transaction to have STO in Algorithm 2. Furthermore, all these conditions may be evaluated by analyzing the local DAG before b' is committed; therefore, early finality for Type β transactions is achievable. We provide complete formal proofs in Lemma A.3.

5.4 Type γ : Coordinated Cross-Shard Transactions

Coordinated operations across multiple shards are a fundamental requirement in distributed databases. Atomic and serializable tuples of transactions enable consistent writes across shards. Combined with Type α, β transactions, these three transaction types are sufficient to express the core operations required by most distributed database workloads.

A Type γ transaction is a specialized tuple of Type α/β sub-transactions that are *atomic* and *serializable as a tuple*. As mentioned in §5.1, we will discuss Type γ transactions as a pair of sub-transactions.

Atomicity is guaranteed if all sub-transactions are eventually executed; this can be achieved by having both sub-transactions include each other as part of its metadata. Therefore, if one is part of a block, the other will be known by all nodes and eventually be incorporated into the DAG and committed as well.

The challenge lies in ensuring that Type γ sub-transactions are *serializable as a pair*. We illustrate this with an example: Consider a Type γ transaction that swaps the values of two keys $k_j^1 \mapsto \text{apple}$ and $k_i^1 \mapsto \text{orange}$, where $k_j^1 \in k_j, k_i^1 \in k_i$ and $k_i \neq k_j$. In this particular instance, the Type γ transaction

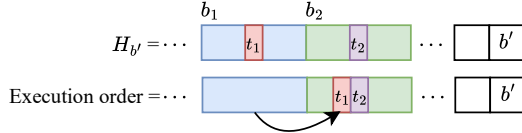


Figure 8: This figure shows how a Type γ transaction (t_1, t_2) affects the execution order.

comprises two Type β sub-transactions:

1. Sub-transaction 1: Read k_j^1 and write it into k_i^1 .
2. Sub-transaction 2: Read k_i^1 and write it into k_j^1 .

The desired outcome should be $k_j^1 \mapsto \text{orange}, k_i^1 \mapsto \text{apple}$. However, as the sub-transactions execute independently and sequentially, the result will be both keys having identical values. This is the case even if both transactions are adjacent in the sorted causal history of a leader that commits them both. Furthermore, a third transaction (such as writing *mango* to k_j^1) may be ordered between the two sub-transactions, causing the final outcome to deviate from the desired result.

Therefore, the sub-transactions that constitute a Type γ transaction must be executed atomically at the *same time* for the desired outcome to be achievable.

5.4.1 Ordering and Executing Type γ Sub-Transactions

Enforcing that both sub-transactions of a Type γ transaction are executed together is non-trivial, especially since both components of a Type γ transaction may exist in different blocks, potentially in different rounds and within different committed leaders' causal histories.

Type α and β transactions follow the commitment and thereby execution procedure described in Definition 4.1, while Type γ transaction *executions deviate slightly*. Consider two sub-transactions that constitute a Type γ transaction, $t_1 \in b_i^{r_1}$ and $t_2 \in b_j^{r_2}$, where $b_i^{r_1} \neq b_j^{r_2}$. We consider 3 cases:

- Consider $r_1 = r_2$ and $b_i^{r_1}, b_j^{r_2}$ are committed by the same leader. Suppose $b_i^{r_1}$ is ordered before $b_j^{r_2}$ (per Definition 4.1). t_1 will not be executed with the other transactions in $b_i^{r_1}$. Instead, t_1 is executed concurrently with t_2 in $b_j^{r_2}$. We illustrate this example in Fig. 8.
- Consider $r_1 < r_2$ and $b_i^{r_1}, b_j^{r_2}$ are committed by the same leader. Since $b_i^{r_1}$ is from an earlier round, it is ordered before $b_j^{r_2}$ in the leader's sorted causal history. Therefore, similar to the previous case, t_1 is executed together with t_2 in $b_j^{r_2}$.
- Consider $b_i^{r_1}, b_j^{r_2}$ are committed by different leaders (with $b_i^{r_1}$ committed earlier). Transaction t_1 will not execute until t_2 is committed, and t_1 will similarly be executed together with t_2 in $b_j^{r_2}$.

Essentially, although sub-transactions may exist in distinct blocks, they are reordered such that they execute together as if both exist in *one* of the 2 blocks. For illustration, we denote that block as the **prime** block and the corresponding transaction (that exists physically in the block) as the prime sub-transaction. The other block/transaction is likewise referred to as the non-prime block/sub-transaction.

A challenge of achieving early finality for Type γ transactions is determining which transaction is *prime*. This is mainly because it requires us to witness both blocks and identify which leader's causal history they are part of.

Lemonshark's technique to achieve early finality for Type γ transactions is straightforward: if we can guarantee that both sub-transactions are part of the same leader's causal history, we can determine which sub-transaction is prime. Subsequently, from the position of the prime sub-transaction, we evaluate if STO for both sub-transactions can be achieved before it is committed.

5.4.2 Same Round, Same Leader

We now describe how early finality for Type γ transactions may be achieved per the first case described in §5.4.1. Consider two sub-transactions that constitute a Type γ transaction, t_1 and t_2 , existing in blocks $b_i^{r_1}$ and $b_j^{r_2}$ respectively.

Same Leader's Causal History The requirement that both blocks exist within the same leader's causal history is readily satisfied. This condition is trivially met when a leader in round $\max(r_1, r_2) + 1$ contains both blocks within its causal history. Conversely, when no leader from a round prior to $\max(r_1, r_2) + 1$ has either block in its causal history, the persistence of both blocks in round $\max(r_1, r_2) + 1$ ensures their eventual inclusion in the same committed leader's (b') causal history. We establish this property in Proposition A.7.

Suppose b' is the leader that ultimately commits both blocks. Under the assumption that $r_1 = r_2$, suppose that $b_i^{r_1}$ precedes $b_j^{r_2}$ in the ordering of $H_{b'}$, t_2 then constitutes the prime sub-transaction.

Condition for STO Firstly, we require that t_1 and t_2 be evaluated to have STO independently—treating each as an autonomous Type α or β transaction. When the non-prime sub-transaction t_1 achieves STO independently, this indicates that all uncommitted blocks capable of influencing its outcome that exist in rounds $< r_1$ are contained within $b_i^{r_1}$.

Similarly, satisfying the leader-checks ensures that blocks in the subsequent round $r_1 + 1 = r_2 + 1$ will not supersede them during execution according to our sorting procedure established in Definition 4.1. Given that both $b_i^{r_1}$ and $b_j^{r_2}$ are elements of $H_{b'}$, this guarantees that t_1 would maintain STO if it were the sole transaction in $b_j^{r_2}$.

When no additional transactions exist in $b_i^{r_1}$ and $b_j^{r_2}$, this condition alone suffices for STO evaluation of t_1 and t_2 . However, given that this scenario is not frequently realized, we must account for the potential influence of other transactions in $b_i^{r_1}$ and those preceding $t_2 \in b_j^{r_2}$ on the outcome of t_1 . To address this complexity, we impose a comprehensive constraint: all remaining transactions in both $b_i^{r_1}$ and $b_j^{r_2}$ must have STO. This requirement enables the deterministic assessment of their impact on t_1 in advance, enabling us to produce STOs for both t_1 and t_2 that match the execution prefix of t_1 and t_2 with respect to b' .

Given that every transaction within $b_i^{r_1}$ and $b_j^{r_2}$ can be deter-

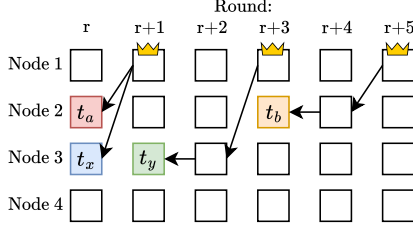


Figure 9: How speculation can aid in reducing latencies for dependent transactions. Suppose the pairs t_a, t_b and t_x, t_y are dependent. However, we allow t_y to be submitted earlier along with a speculated result for t_x . t_y execution is contingent on whether that a speculated result is equivalent to the finalized outcome of t_x .

mined to possess STO in round $\max(r_1, r_2) + 1$ before commitment, early finality remains achievable for Type γ transactions. The complete formal proof is presented in Lemma A.4.

We relegate the more intricate case where $r_1 \neq r_2$ to the appendix (Lemma A.5). Nevertheless, the underlying principle is intuitive: if we can reduce the problem to the one described in this section, then early finality is achievable as well.

5.4.3 Different Leaders, Delay List

As highlighted by the third case specified in §5.4.1—it is entirely possible that $b_i^{r_1}$ and $b_j^{r_2}$ are committed by two distinct leaders. In this particular case, we are unable to evaluate the TO of the non-prime sub-transaction (suppose it is $t_1 \in b_i^{r_1}$) until the prime counterpart is committed. This inability to evaluate the TO of t_1 makes it impossible to derive the outcome of subsequent transactions from shard k_i that read or write to the same key that t_1 modifies. This naturally implies that STO would be impossible to derive as well.

We handle this constraint with a blacklisting approach: by adding t_1 in the Delay List for round r_1 (DL_{r_1}). The delay list DL_r contains transactions from rounds up to r that are appended in this manner. If a transaction in a block from round r reads or writes to the same key modified by a transaction that exists in DL_r , that transaction automatically becomes ineligible to achieve STO. To ensure that the delay list is comprehensive, we show in Proposition A.8 that when a node deems that a transaction were to fulfill all other requirements for STO, its view of the Delay List must include all transactions that may affect it.

A Type γ sub-transaction is only removed from DL if both sub-transactions are committed, or when the prime sub-transaction is evaluated to have STO. We elaborate on how the latter is possible in Lemma A.5, where t_1 and t_2 are part of the same leader’s causal history but in different rounds.

An unfortunate consequence of the Delay List mechanism is that if t_1 is committed earlier, we are delaying its execution until t_2 is committed, possibly incurring rounds of additional delay. Nevertheless, because Type γ transactions are assumed to be relatively rare (due to how the key-space is partitioned into shards), we expect this issue to have only a negligible impact in practice.

6 Pipelined Dependent Transactions

Orthogonal to our main insight in §5, we observe that clients with dependent transactions (where subsequent transactions rely on previous transaction outcomes) traditionally face multiplicative latency penalties. By providing tentative, speculated results earlier, subsequent transactions may be included in the DAG sooner (Fig. 9). When speculation succeeds, this approach achieves lower latencies; when incorrect, transactions are aborted and latency reverts to the baseline case. We provide a more detailed analysis, experimental results, and the integration with Lemonshark’s core contributions in Appendix F.

7 Implementation

We implement Asynchronous Bullshark by forking Bullshark’s [3] open source repository, which itself forks from the Narwhal project [7] and utilizes Narwhal’s DAG structure for its consensus core.

We then implement Lemonshark on top of our Asynchronous Bullshark implementation to interpret the DAG differently for early finality. The implementation is written in Rust and uses `tokio` [9] for asynchronous networking. `ed25519-dalek` [4] is used as the cryptographic library and RocksDB [8] for persistent storage of the DAG.

Baselines: We compare Lemonshark explicitly against Bullshark [27] as it is the only state-of-the-art asynchronous DAG-BFT with a complete open-source implementation. We exclude Tusk [22] from comparison as Bullshark’s greater leader election frequency yields 33% lower latencies. We also exclude Agreement-on-Common-Set asynchronous BFT protocols like Dumbo [37], as Tusk demonstrates substantial (20×) latency improvements over these approaches. Uncertified DAG BFT protocols either lack practical implementations [32] or operate under different network synchrony assumptions [12], which makes direct comparison with Lemonshark inappropriate.

8 Evaluation

In theory, early finality in Lemonshark should improve transaction latency while maintaining throughput and scalability of a DAG-BFT protocol. We now evaluate Lemonshark to empirically demonstrate the benefit of early finality. In particular, we aim to answer the following questions:

- Can Lemonshark achieve consistently lower latency through early finality, while maintaining high throughput and scalability?
- Can Lemonshark sustain its performance even for cross-shard transactions?
- Can Lemonshark maintain its latency benefit even in the presence of failures?

We use two types of latencies in the evaluation. **Consensus latency** refers to the time taken for a block to be finalized after its reliable broadcast. **End-to-end (E2E) latency** refers to the duration for a transaction within a block to be finalized after it has been generated by the client. For Type γ transactions,

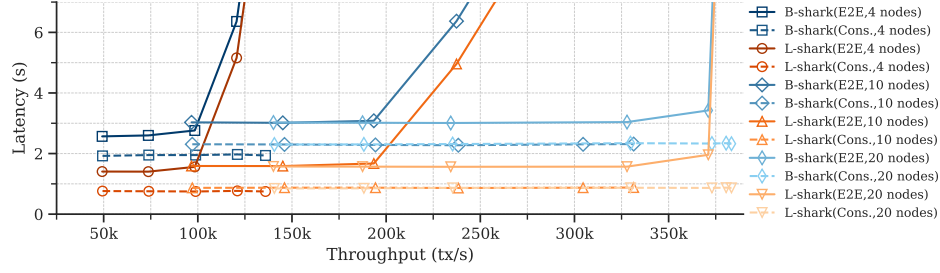


Figure 10: Performance of Lemonshark with Type α transactions vs Bullshark with no faults, varying the number of nodes.

both E2E/consensus latencies are recorded with respect to the latest submitted sub-transaction onto the DAG.

Experimental Setup: We deploy our testbed on AWS, utilizing m5.8xlarge [2] instances distributed across 5 regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1). Each machine has 10Gbps bandwidth, 32 vCPUs running at 3.1GHz, and 128GiB memory. These machines are selected as they mirror those deployed by production blockchains and represent commodity servers.

As described in §5.1, shard-rotation is deterministic and public, but clients should broadcast transactions to all nodes. Following previous works [11, 22, 27], our clients connect locally to each instance and continuously send streams of 512B “nop” transactions. This approach isolates consensus performance from two orthogonal factors: (1) client geolocation, which would otherwise introduce variable network delays based on client proximity to instances, and (2) transaction execution overhead, which would confound measurements of consensus latency². To evaluate cross-shard operations, we mark each block’s meta at dissemination to denote transaction types it carries (e.g., whether it contains a set of type β transactions that read from other shards).

Narwhal proposes scaling dissemination by utilizing an additional “worker-layer” that performs the first step of reliable broadcast in batches. Blocks in the DAG then contain hashes representing batches. In our experiments, we utilize batch sizes of 500KB and block sizes of 1000B³.

We utilize a *leader-timeout* value of 5 seconds; if a steady leader should exist in a round, each node waits that duration before proceeding without the leader’s block in its local DAG. As discussed in §3.1.1, byzantine nodes are unable to equivocate due to the reliable broadcast primitive. Therefore, in our experiments, we simulate only *crash-faults*. We also deviate from previous experimental methodologies [22, 27], normalizing failure behavior by randomly selecting faulty nodes and randomizing steady leader selection. We elaborate in Appendix E on why this approach is fairer. We also elaborate in Appendix D how missing blocks caused by crash faults are

² We measured a maximum ~ 300 ms network latency between the most distant instance pair in our deployment. Even accounting for this maximal penalty, Lemonshark achieves superior end-to-end latencies in all experiments. ³ Since each hash is stored in a 32B digest, a 1000B block represents approximately 32K transactions

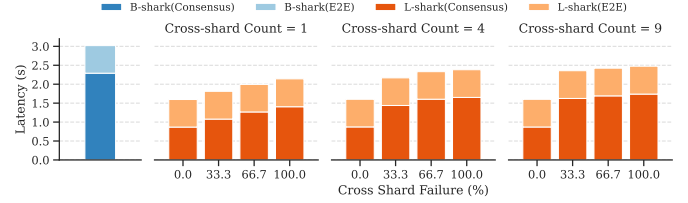


Figure 11: Performance of Lemonshark with Type β transactions, while varying the number of cross-shard reads per block, and failure rates.

identified by honest nodes. All measurements represent the average of 3 independent runs of 3 minutes.

8.1 Single Shard Transactions

Lemonshark’s latency is optimal when all transactions are Type α . As such, we compare that performance with Bullshark with no crash-faults present. Since every non-leader block in Lemonshark should qualify for early finality after witnessing sufficient blocks in the subsequent round, consensus latency for all blocks approaches optimal at approximately 65% lower than Bullshark (Fig. 10). This performance is sustained even when we increase client transaction rates and committee sizes. Eventually, client transaction rates exceed consensus capacity, causing queues to form indefinitely and latencies to spike unavoidably.

For the remainder of our experiments, we utilize a baseline transaction rate of 100K-tx/s and a committee size of 10. This represents a moderate load that effectively stresses the consensus mechanism while remaining within capacity.

8.2 Cross-Shard Transactions

Since Type γ transactions consist of pairs of Type α/β transactions, their performance closely correlates with that of their constituent sub-transactions in failure-free scenarios. Therefore, we evaluate cross-shard behavior by simulating Type β transactions and γ sub-transactions. We configure 50%⁴ of all blocks to contain cross-shard transactions that either read from a number of shards equal to “Cross-shard Count” or have sub-transactions distributed across that many shards. When a block in round r contains cross-shard transactions, we randomly determine the number of shards it interacts with (uniformly from 0 to “Cross-shard Count”) for either reads or sub-transaction placement. For each shard from which it reads, “Cross Shard Failure” denotes the probability that the

⁴ We vary this proportion in Appendix E.3

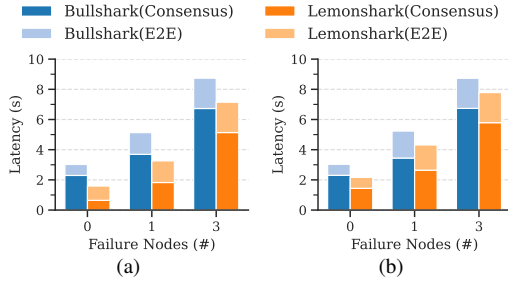


Figure 12: Performance of (a) Type α (b) Type β/γ (with moderate amount of cross-shard activity, (Cross-shard Count = 4, Cross Shard = 33% while varying the number of faults).

read key is modified by another block in round r , or that the companion sub-transactions do not exist in the same round. d

Given the relatively synchronous nature of the AWS network, the primary factor preventing a Type β transaction t (from a block in round r reading from k_j) from obtaining STO is not pointing to blocks from the previous round (§5.3.1) or failing the Leader-check (§5.3.3), but rather when b_j^r modifies the read value (§5.3.2). However, when b_j^r commits before t , we can determine its effect on the read value and evaluate STO for t . This scenario explains why, even when Type β transactions are abundant and “Cross-Shard Failure” rates are high, our consensus latencies remain approximately 25% lower (Fig. 11) than those of Bullshark. Even when all transactions are cross-shard, we maintain approximately 18% consensus latency reduction when “Cross-shard Count = 4” (the details are in Appendix E.3).

8.3 Performance under Failures

We vary the number of crash-faults to evaluate their impact on consensus latency. When $f = 1$, Lemonshark’s consensus latency improvement decreased from approximately 65% to approximately 50%, and further declined to approximately 24% when $f = 3$. As demonstrated in Fig. 12, Type β/γ transactions exhibited similar performance degradation: approximately 23% improvement when $f = 1$ and approximately 14% when $f = 3$. This behavior is expected, as commitment disruption reduces the likelihood of the scenario described in §8.2. For Type β transactions, t and b_j^r are often committed together by a fallback leader instead. Nevertheless, Lemonshark consistently achieves lower latencies than Bullshark across all failure scenarios.

8.3.1 Missing Blocks In-Charge of a Shard

Since only a single node may write to a particular shard per round, if the node in-charge of the shard is faulty, transactions designated to that shard will be delayed till an honest becomes in-charge of that shard.

This represents a fundamental trade-off in Lemonshark’s sharding design; if a transaction is submitted while the node in-charge of it is unavailable, that transaction will incur an additional delay. **Those unfortunate transactions perform slightly worse** than Bullshark, with an average increase of approximately 500ms (approximately 12%) in end-to-end latency when $f = 1$, and approximately 1500ms (approximately

17%) when $f = 3$.

9 Related Work

Traditional Asynchronous BFT Traditional DAG-free asynchronous BFT protocols [24, 35, 37, 39] utilize either the BKR [14, 15] or CKPS [18] paradigm, both resulting in BAB protocols with expected $O(\log n)$ time complexity. DAG-Rider [31] established the current standard for asynchronous BAB protocols by achieving constant expected time complexity through reliable broadcast primitives [17].

DAG-BFT with Weaker Network Assumptions Several descendants [11, 27, 45] of early DAG-BFT protocols [13, 22, 31] reduce commitment latency through leader-focused strategies, such as increasing leader count or introducing faster commitment rules. These improvements come at the cost of reduced resilience to asynchrony [11, 45]. Autobahn [28] addresses the inefficiency of lock-step dissemination in DAG-BFT but employs PBFT [19]-style commitment, weakening network assumptions to partial synchrony. In contrast, Lemonshark preserves strong asynchrony assumptions, aligning with DAG-Rider [31] and Tusk [22]. Since Lemonshark builds upon Bullshark, our insights extend to these related protocols.

Uncertified DAG-BFT Recent work has introduced “uncertified DAGs” [32, 38] to reduce DAG-BFT latency by replacing expensive reliable broadcast primitives with best-effort dissemination. However, this introduces a synchronization overhead on the critical path [11, 28]. This overhead can outweigh the latency savings, resulting in significant performance degradation under minor message loss.

Fast Finality in uncertified DAG-BFT Other uncertified DAG-BFT protocols, such as Mysticeti-FPC [12], employ explicit voting to ensure conflict-free transaction ordering for distinct keys, allowing certain transactions to achieve finality faster. However, this approach differs fundamentally from Lemonshark: Mysticeti-FPC requires explicit coordination, whereas Lemonshark implicitly determines whether early finality conditions are satisfied through local DAG inspection. For simplicity, Lemonshark evaluates SBO recursively starting from the earliest uncommitted block, though the protocol can be extended to evaluate STO exclusively (see Appendix C). Moreover, while Mysticeti-FPC trades network resilience for performance, it still inherits the synchronization issues fundamental to all uncertified DAG-BFT protocols.

10 Conclusion

In this work, we identify latency disparities between leader and non-leader blocks in DAG-BFT protocols and present Lemonshark, an asynchronous DAG-BFT protocol that enables *early finality* through key-space sharding. Early finality allows non-leader blocks to return finalized results before commitment when specific conditions are satisfied, enabling them to achieve optimal latencies previously exclusive to leader blocks.

References

- [1] Accolade gathers \$202 million for new blockchain fund of funds as crypto market recovers. <https://arc.net/1/quote/gvxzbiuv>.
- [2] Amazon ec2 instance types. <https://aws.amazon.com/ec2/instance-types/m5/>.
- [3] Bullshark github implementation. <https://github.com/asonnino/narwhal/tree/bullshark-fallback>.
- [4] Dalek elliptic curve cryptography library. <https://github.com/dalek-cryptography/curve25519-dalek>.
- [5] European central bank: The digital euro. https://www.ecb.europa.eu/euro/digital_euro/html/index.en.html.
- [6] Janus henderson to follow blackrock and fidelity into tokenisation. <https://arc.net/1/quote/srjmeosq>.
- [7] Narwhal github implementation. <https://github.com/facebookresearch/narwhal>.
- [8] Rocksdb, a persistent key-value store. <https://rocksdb.org/>.
- [9] Tokio: An asynchronous runtime for the rust programming language. <https://tokio.rs/>.
- [10] ABRAHAM, I., MALKHI, D., AND SPIEGELMAN, A. Asymptotically optimal validated asynchronous byzantine agreement. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019).
- [11] ARUN, B., LI, Z., SURI-PAYER, F., DAS, S., AND SPIEGELMAN, A. Shoal++: High throughput dag bft can be fast! *ArXiv abs/2405.20488* (2024).
- [12] BABEL, K., CHURSIN, A., DANEZIS, G., KICHIDIS, A., KOKORIS-KOGIAS, L., KOSHY, A., SONNINO, A., AND TIAN, M. Mysticeti: Reaching the latency limits with uncertified dags. *Proceedings 2025 Network and Distributed System Security Symposium* (2023).
- [13] BAIRD, L. The swirls hashgraph consensus algorithm: fair, fast, byzantine fault tolerance. *Swirls technical report TR-2016-01* (2016).
- [14] BEN-OR, M. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada, 1983), PODC '83, Association for Computing Machinery, pp. 27–30.
- [15] BEN-OR, M., KELMER, B., AND RABIN, T. Asynchronous secure computations with optimal resilience (extended abstract).
- [16] BONEH, D., LYNN, B., AND SHACHAM, H. Short signatures from the weil pairing. *Journal of Cryptology* 17 (2001), 297–319.
- [17] BRACHA, G. Asynchronous byzantine agreement protocols. *Inf. Comput.* 75 (1987), 130–143.
- [18] CACHIN, C., KURSAWE, K., PETZOLD, F., AND SHOUP, V. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference* (2001).
- [19] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, Feb. 1999), OSDI '99, USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.
- [20] CLARK, J., DONALDSON, A. F., WICKERSON, J., AND RIGGER, M. Validating database system isolation level implementations with version certificate recovery. *Proceedings of the Nineteenth European Conference on Computer Systems* (2024).
- [21] CORREIA, M. P., NEVES, N. F., AND VERÍSSIMO, P. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. 82–96.
- [22] DANEZIS, G., KOKORIS-KOGIAS, L., SONNINO, A., AND SPIEGELMAN, A. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems* (New York, NY, USA, 2022), EuroSys '22, Association for Computing Machinery, p. 34–50.
- [23] DOLEV, D., FISCHER, M. J., FOWLER, R. J., LYNCH, N. A., AND STRONG, H. R. An efficient algorithm for byzantine agreement without authentication. *Inf. Control.* 52 (1982), 257–274.
- [24] DUAN, S., REITER, M. K., AND ZHANG, H. Beat: Asynchronous bft made practical. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018).
- [25] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM* 32, 2 (Apr. 1985), 374–382.
- [26] GAGOL, A., LESNIAK, D., STRASZAK, D., AND SWIETEK, M. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* (2019).
- [27] GIRIDHARAN, N., KOKORIS-KOGIAS, L., SONNINO, A., AND SPIEGELMAN, A. Bullshark: Dag bft protocols made practical. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022).
- [28] GIRIDHARAN, N., SURI-PAYER, F., ABRAHAM, I., ALVISI, L., AND CROOKS, N. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (New York, NY, USA, 2024), SOSOP '24, Association for Computing Machinery, p. 1–23.
- [29] HAN, R., YU, J., AND ZHANG, R. Analysing and improving shard allocation protocols for sharded blockchains. *Proceedings of the 4th ACM Conference on Advances in Financial Technologies* (2022).
- [30] KAHN, A. B. Topological sorting of large networks. *Communications of the ACM* 5 (1962), 558 – 562.
- [31] KEIDAR, I., KOKORIS-KOGIAS, E., NAOR, O., AND SPIEGELMAN, A. All You Need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy, 2021), PODC '21, Association for Computing Machinery, pp. 165–175.
- [32] KEIDAR, I., NAOR, O., POUPKO, O., AND SHAPIRO, E. Y. Cordial miners: Fast and efficient consensus for every eventuality. In *International Symposium on Distributed Computing* (2022).
- [33] KINGSBURY, K., AND ALVARO, P. Elle: Inferring isolation anomalies from experimental observations. *ArXiv abs/2003.10554* (2020).
- [34] LIBERT, B., JOYE, M., AND YUNG, M. Born and raised distributively: fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Proceedings of the 2014 ACM symposium on Principles of distributed computing* (2014).
- [35] LIU, C., DUAN, S., AND ZHANG, H. Epic: Efficient asynchronous bft with adaptive security. *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2020), 437–451.
- [36] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011).
- [37] LU, Y., LU, Z., TANG, Q., AND WANG, G. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. *Proceedings of the 39th Symposium on Principles of Distributed Computing* (2020).
- [38] MALKHI, D., STATHAKOPOULOU, C., AND YIN, M. Bbca-chain: One-message, low latency bft consensus on a dag. *ArXiv abs/2310.06335* (2023).
- [39] MILLER, A., XIA, Y., CROMAN, K., SHI, E., AND SONG, D. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria, 2016), CCS '16, Association for Computing Machinery, pp. 31–42.

- [40] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).
- [41] OKANAMI, N., NAKAMURA, R., AND NISHIDE, T. Load balancing for sharded blockchains. In *Financial Cryptography Workshops* (2020).
- [42] PETERSEN, K., SPREITZER, M., TERRY, D. B., AND THEIMER, M. Bayou: replicated database services for world-wide applications.
- [43] SHEN, W., CUI, Y., SEN, S., ANGEL, S., AND MU, S. Mako: Speculative distributed transactions with geo-replication. In *USENIX Symposium on Operating Systems Design and Implementation* (2025).
- [44] SHOUP, V. Practical threshold signatures. In *International Conference on the Theory and Application of Cryptographic Techniques* (2000).
- [45] SPIEGELMAN, A., AURN, B., GELASHVILI, R., AND LI, Z. Shoal: Improving dag-bft latency and robustness. *ArXiv abs/2306.03058* (2023).
- [46] SPIEGELMAN, A., GIRIDHARAN, N., SONNINO, A., AND KOKORIS-KOGIAS, L. Bullshark: The partially synchronous version. *ArXiv abs/2209.05633* (2022).
- [47] TAN, C., ZHAO, C., MU, S., AND WALFISH, M. Cobra: Making transactional key-value stores verifiably serializable. In *USENIX Symposium on Operating Systems Design and Implementation* (2020).

A Definitions and Proofs

A.1 Definitions for Bullshark

This subsection describes the Bullshark consensus engine. We utilize definitions to illustrate better how Lemonshark works in the later sections.

A.1.1 Blocks

Definition A.1 (Rounds and Waves). *The protocol progresses in virtual rounds; For every round. Beginning from round 1, every 4 rounds constitute a wave; round 1-4 belongs to wave 1, round 5-8 to round 2, and so on.*

In each round (r), a node p_i may call upon the Reliable Broadcast Primitive to disseminate a message ($rbc(m, p_i, r)$) where m is the message. When a node receives that message, it calls $deliver(m, p_i, r)$. The Reliable Broadcast primitive guarantees the following properties:

- **Agreement:** If an honest party calls $deliver(m, p_i, r)$, then eventually all honest nodes will call it too.
- **Integrity:** In any round, an honest party may only call $deliver(m, p_i, r)$ at most once for p_i .
- **Validity:** If an honest party p_i calls $rbc(m, p_i, r)$, eventually all honest nodes will call $deliver(m, p_i, r)$.

In this work, we imagine a 2-phased reliable broadcast primitive akin to Bracha’s reliable broadcast [17].

Definition A.2 (Blocks and Pointers). *A block b is the result of a message being delivered $deliver(m, p_i, r)$; we say p_i produced the block b in round r . The message m includes a set of transactions $b = [t_1, \dots, t_i, \dots, t_p]$ ordered by p_i , as well as a set of at least $2f + 1$ blocks from $r - 1$. We say b has pointers to any block within that set.*

In Bullshark, blocks from r might have direct pointers to blocks in $r' : r' < r - 1$. These pointers are called “weak-links”. However, in this work, we disregard this optimization and focus solely on what they refer to as “strong-links”; or pointers to the immediate previous round $r - 1$.

It is also clear how the blocks and their pointers intrinsically form a DAG, where the blocks are the vertices and the pointers represent the edges.

Definition A.3 (Block Path). *For a block b' in round r' and a block b in round $r : r' > r$, b' has a path to b if 1) b' has a pointer to b , or 2) there exist blocks in rounds $r' - 1$ to $r + 1$ that b' has pointers recursively to b .*

A.1.2 Leaders and Votes

In Bullshark, there exist two classes of leaders: Stable leaders for when the network is relatively synchronous, and fallback leaders when there exists byzantine interference or asynchrony.

Definition A.4 (Stable Leader). *A Stable leader is a pseudonym that is deterministically given to a block in the first and third rounds of any wave.*

Stable leaders are typically assigned in a round-robin manner.

Definition A.5 (Fallback Leader). *A fallback leader is a pseudonym randomly given to a block in the first round of any wave. This allocation is only revealed at the end of the fourth round of the wave, utilizing a global perfect coin.*

Definition A.6 (Raw Causal history of a block). *The raw causal history of a particular block b is the blocks to which it has a path.*

Definition A.7 (Stable Vote). *For a particular wave $w : w > 1$, in the raw causal history of the block produced by p_i in the first round of the wave: if it shows either the 2nd Stable leader in wave $w - 1$ or the Fallback leader is committed, then paths by blocks produced in the 2nd and last rounds in w by the same node are considered stable votes.*

Definition A.8 (Fallback Vote). *For a particular wave $w : w > 1$. In the raw causal history of the block produced by p_i in the first round of the wave: if it shows neither the 2nd Stable leader in wave $w - 1$ or the Fallback leader is committed. Paths by the block produced by p_i in the last round of w to the Fallback leader is considered a fallback vote.*

In Bullshark, a leader that has garnered sufficient votes is considered **Committed**. By quorum intersection, it’s clear to see that only a single leader type may be committed for the first round of a wave.

Definition A.9 (Committed Leader). *A leader is “Committed” once it has received sufficient votes; a fallback leader that has received at least $2f + 1$ fallback votes, or a stable leader that has received at least $2f + 1$ stable votes in the same wave. A leader is also “Committed” if it’s in another committed leader’s causal history, and has obtained sufficient votes: $\geq f + 1$ of the appropriate vote type with less than $f + 1$ votes of the other vote type present.*

A.1.3 Causal Histories

Once a leader is committed, the blocks it has paths to are ordered in some deterministic ordering, where transactions are ordered at the block level. To prevent duplicate execution, blocks that the previous committed leader has paths to are excluded. As such, we define:

Definition A.10 ((Sorted) Causal History of a Block). *For a block b , its causal history is the set of nodes that are part of a sub-DAG, where b is the root, exclusive of the blocks that are in the previous known committed leader’s causal history. Its sorted causal history is obtained by applying Kahn’s algorithm to the sub-DAG and reversing the list. where ties (blocks of the same round) are broken deterministically. We denote this list as $H_b = [\dots, b]$.*

The causal history of any block excludes blocks that are already committed, as those blocks cannot be committed again and are safe to ignore. In contrast to previous works that accept any deterministic method, we specify a particular ordering procedure. This choice is both intuitive (older blocks execute first, in round-by-round order) and critical to

Lemonshark, ensuring that older blocks never precede newer blocks. We do not impose constraints on how blocks from the same round are ordered, provided the ordering remains deterministic. We illustrate this concept in Fig. 3.

Once a leader commits, we commit and execute transactions within blocks of its sorted causal history sequentially, proceeding block by block. Consequently, non-leader blocks are *committed* if and only if they exist in the sorted causal history of a committed leader block, where any non-leader block may be included in at most one committed leader's causal history. For a committed leader b' , we say it *commits* $b : b \in H_{b'}$ as well as all transactions in b . Given the strict monotonic ordering between committed leaders, for leaders b' and b'' that are committed consecutively, all elements in $H_{b'}$ are committed before those in $H_{b''}$.

Definition A.11 (Commitment of a non-leader block). *For a non-leader block b , it is committed if it is in the causal history of a committed leader block $b' : b \in H_{b'}$ and b' has sufficient votes. We say b and its transactions are committed by b' in the order of $H_{b'}$.*

Definition A.12 (Commitment ordering). *For two subsequently committed leaders b', b'' from rounds r', r'' such that $r' < r''$ where $H_{b'} = [b_1, b_2, b']$, $H_{b''} = [b_x, b_y, b'']$. We say b' and elements in $H_{b'}$ are committed before b'' and elements in $H_{b''}$. Similarly in $H_{b'}$ we say b_1 is committed before b_2 .*

A.1.4 Transaction/Block Outcomes

Definition A.13 (Key-spaces and Transactions). *Let K represent the key-space of a database, and a transaction as a unit of work that modifies a key $k \in K$ in the database.*

Definition A.14 (Transaction Outcome (TO)). *For a block $b = [t_1, \dots, t_i, \dots, t_p]$, the transaction outcome (TO) of t_i is the outcome of t_i when executing transactions in the following order: $H_b[-1] + [t_1, \dots, t_i]$, where $H_b[-1]$ is the prefix of H_b exclusive of b .*

Definition A.15 (Block Outcome (BO)). *For a block b and its Sorted Causal History H_b , the Block Outcome (BO) of b is the execution results of all transactions in b after executing the blocks in the order of H_b .*

It is key to note that a block's causal history changes based on which leader a node believes is the latest that has been committed. Therefore, a node's view on a block's sorted causal history, as well as the TO of the transactions within the block might not be static.

Definition A.16 (Execution Prefix (Block)). *For blocks $b', b : b = [t_1, \dots, t_i, \dots, t_p]$, where $b \in H_{b'}$. The execution prefix $b' \langle b \rangle$ is the outcome of the transactions in b when executing the prefix of $H_{b'}$ up to and including b (i.e., $H_{b'}[0 : \text{index}(b)]$ or $H_{b'}[0 : \text{index}(b) - 1] + [t_1, \dots, t_i, \dots, t_p]$). We describe this as the execution prefix of b with respect to b' .*

Definition A.17 (Execution Prefix (Transactions)). *For blocks $b', b : b = [t_1, \dots, t_i, \dots, t_p]$, where $b \in H_{b'}$. The execution prefix $b' \langle b(t_i) \rangle$ is outcome of transaction t_i when executing the prefix of $H_{b'}$ right before b and $[t_1, \dots, t_i]$ (i.e., $H_{b'}[0 :$*

$\text{index}(b) - 1] + [t_1, \dots, t_i]$). We describe this as the execution prefix of t_i with respect to b' .

When the leader block b' (where $b \in H_{b'}$) obtains sufficient votes to commit in round r' , the execution prefix of each block/transaction in $H_{b'}$ with respect to b' becomes the *finalized immutable outcome*.

Definition A.18 (Safe Transaction Outcome (STO)). *For a particular transaction $t_i \in b$, we say that its transaction outcome is safe if for a committed leader block $b' : b \in H_{b'}$, the transaction outcome of t_i is equivalent to the execution prefix $b' \langle b(t_i) \rangle$.*

Definition A.19 (Safe Transaction Outcome (SBO)). *If all transactions within a block b have STO, we say the block has a safe block outcome (SBO).*

Definition A.20 (Early Finality). *For a non-leader block b , early finality is achieved for b if the SBO of b may be determined before b is determined to be committed.*

Definition A.21 (Block Persistence). *We say that a block b in round r persists in round $r' : r' > r$, if for any set of $2f + 1$ blocks in round r' , at least one block in the set has a path to b .*

Proposition A.1. *A block b in round r has greater than f blocks in round $r + 1$ pointing to it iff it persists in round $r + 1$.*

Proof. By quorum intersection. If there are at least $f + 1$ blocks in round $r + 1$ which have a path to b , then clearly any subset of size $2f + 1$ blocks must contain at least one such block. If b persists, assume for the sake of contradiction that at most f blocks point to it. Take the set of all blocks that do not have a path to b . This set clearly has size at least $2f + 1$, contradicting that b persists. \square

The notion of Block persistence is that if a block has sufficient pointers to it at some round, then it will definitely be included in some committed leader's causal history

A.1.5 Impossibility of Early Finality in Bullshark

In Bullshark, there is no restriction on which transactions may be included in which block. As such, a block may include transactions that touch on all keys in K . Therefore, in our proof, we demonstrate that it's precisely because of this general approach to transaction block inclusion that leads to Bullshark being unable to achieve early finality.

Proposition A.2. *Due to network asynchrony, there may exist at least f blocks from round r that do not persist in $r + 1$.*

Proof. Suppose at round r , there exists a 2 partitions of nodes: π_1, π_2 such that $\pi_1 \cap \pi_2 = \emptyset, \pi_1 \cup \pi_2 = \Pi, |\pi_1| = 2f + 1$. Its possible that the blocks in π_1 only learn of blocks in its own partition. As a result, its blocks in $r + 1$ only point to the blocks produced by nodes in π_1 .

Suppose at this point the partition is lifted. The nodes in π_2 now see all blocks from round r . Each block in π_2 may point

to all blocks created by those in π_2 , but those blocks may only get at most f pointers, and therefore do not persist in $r + 1$. \square

Proposition A.3. *Even under network asynchrony and the presence of Byzantine adversaries, there must exist at least $(3f + 2)/2$ blocks from round r that persist in $r + 1$.*

Proof. Suppose there exist $3f + 1$ blocks in round r , and due to byzantine inaction, there exist only $2f + 1$ blocks in round $r + 1$. There may exist fewer than $3f + 1$ blocks in round r , but with fewer blocks, more blocks persist, as there are fewer options for blocks in round r to point to. To minimize the number of blocks that persist in round $r + 1$, each block in $r + 1$ have exactly $2f + 1$ pointers to blocks in r .

Suppose there exist A blocks from round r that must persist in. To minimize the pointers from blocks in round r that point to the remainder $3f + 1 - A$ blocks, we let all blocks in $r + 1$ point to those blocks in A .

This means each block in $r + 1$ need to point to a remainder of $2f + 1 - A$ blocks in round r .

Since each $2f + 1 - A$ blocks in the remainder $3f + 1 - A$ blocks in r can only have at most f blocks pointing to it. Since there exists $2f + 1$ blocks, we need $\lceil 2f + 1 / f \rceil = 3$ of these sets in the $3f + 1 - A$ blocks. Therefore, we solve the inequality:

$$(3f + 1 - A)(2f + 1 - A) \geq 3 \\ 2f + 1 > A \geq (3f + 2)/2$$

Therefore, $(3f + 2)/2$ is the minimum number of blocks that must persist in $r + 1$. \square

Lemma A.1. *Consider the Bullshark protocol; in the presence of Byzantine adversaries and network asynchrony, it is not possible to determine if a non-leader block b in round r has SBO before it is committed.*

Proof. Suppose b persists in round $r + 1$ and there exists a committed leader b' from r' obtains sufficient votes in round $r' + \omega$ such that $b \in H_{b'}$. We focus on a single transaction $t \in b$, that modifies key $k \in K$.

Consider some block a in round r which contains transactions that modifies every single key in K . By Proposition A.2, it is possible that by the means of an inactive adversary, or manipulated network ordering, a does not persist in round $r + 1$.

It is clear that if a is ordered before b in execution, it will affect the BO of b . Since a does not persist, for any block b' there is guarantee that $a \in H_{b'}$. Suppose that it holds that $a \in H_{b'}$, and for a block b'' it holds that $a \in H_{b''}$.

For early finality, we must evaluate if b has SBO before it is committed.

Now, at round $r' + \omega - 1$, it is not known if b' will get sufficient votes to be committed as leader. If it is committed, then to achieve early finality, we must evaluate if b has SBO by this round. Suppose the SBO evaluation is that a is not committed and executed before transactions of b . Then consider that at round $r' + \omega$ it is revealed that b' is committed as leader, and so a is ordered before b . This conflicts with the evaluated BO for b , and violates correctness. Conversely, suppose the SBO evaluation is that a is committed. Then suppose that at round $r' + \omega$ it is revealed that b' is not committed, and eventually the next leader committed is b'' , where we have that $a \notin H_{b''}$, again violating correctness. As such, it is not possible to correctly decide on the finalized outcome of executing the transactions in b at round $r' + \omega - 1$.

Therefore, early finality is not possible in the presence of Byzantine adversaries and network asynchrony \square

A.2 Lemons shark

Definition A.22 (Sharded Key-space). *For the Key-space $K = \{k_1, k_2, \dots, k_n\}$, let it be partitioned into n distinct shards $k_i = \{k_i^1, k_i^2, \dots\} : \forall i, j \in n, i \neq j, k_i \cap k_j = \emptyset$. We say a block is in-charge of a shard if its transactions only modify keys from that particular shard.*

Definition A.23 (Shard ownership). *A block b from round r and in-charge of a shard $k_i \in K$ is denoted as b_i^r .*

Lemons shark supports three types of transactions, each capable of early finality. For simplicity and as a proof of concept, we present these transaction types and argue that they are sufficient to cover the essential operations in typical database usage, namely, local updates, cross-shard reads, and coordinated atomic actions. The extension to additional transaction types is left for future work. These transaction types are:

- **Type α :** A transaction in a block in-charge of k_i that reads from and writes to k_i .
- **Type β :** A transaction in a block in-charge of k_i that reads from a key in k_j where $j \neq i$ and writes to k_i .
- **Type γ :** An unordered pair of Type α/β sub-transactions that are atomic and pair-wise serializable.

Definition A.24 (Pair-wise serializable). *A pair of sub-transactions is considered Pair-wise serializable if it is executed concurrently, with no other transaction interleaving it.*

Since a Type γ transaction is two separate sub-transactions that may exist in two separate blocks, it's possible that one sub-transaction is committed before the other. For Type γ sub-transactions to be pair-wise serializable, they must be executed together. Therefore, the earlier committed sub-transaction must be delayed. This is achieved by means of a *Delay List*.

Definition A.25 (Delay List (DL)). A Delay list from round r includes an ordered list of transactions belonging to rounds up to r ; we denote this at DL_r . Any transaction t that reads or modifies a key from round r automatically fails to gain STO if there exists a transaction in DL_r that also modifies the same key.

Suppose for a pair of sub-transactions t_1, t_2 that make up a Type γ transaction. If one of the sub-transactions (t_1) is committed before the other, or exists in an earlier round than the other (t_2). The former is placed into the ordered delay list.

It is only removed from the DL once t_2 is committed or has STO. DL_r includes all transactions that might be included in it before and up to round r . Since a transaction that is placed into the delay list has an unknown outcome until the other half is committed. We make the restriction that a transaction t that modifies k may not have STO if there exists a transaction in DL that also modifies k_i .

Since only one block may modify a key k every round, we need not worry about concurrent additions into the DL_r involving the same shard.

Leader check. Here, we discuss in more formal detail the leader check mentioned in §5.2.2

Algorithm A-1 Leader Check

```

1: function LEADER_CHECK( $b^r, k_i$ )
2:   if leader in round  $r+1$  or leader in round  $r+1$  is committed
   then
3:     return true
4:   end if
5:    $w \leftarrow \lfloor r/4 \rfloor + 1$ 
6:    $\text{steady\_ok} \leftarrow$  enough steady votes in wave  $w$ 
7:    $\text{fallback\_ok} \leftarrow$  enough fallback votes in wave  $w$ 
8:   if not  $\text{steady\_ok}$  and not  $\text{fallback\_ok}$  then
9:     return true
10:  end if
11:   $\text{steady\_leader\_check} \leftarrow$  steady leader in  $r+1$  is in charge
    of  $k_i$ 
12:  if  $\text{fallback\_ok}$  and both leaders exist then
13:    return CHECKPATH( $b^r, b_i^{r+1}$ )
14:  end if
15:  if  $\text{steady\_ok}$  and  $\text{steady\_leader\_check}$  then
16:    return CHECKPATH( $b^r, b_i^{r+1}$ )
17:  end if
18:  return true
19: end function
20: function CHECKPATH( $b^r, b_i^{r+1}$ )
21:  return  $b_i^{r+1}$  has path to  $b^r$ 
22: end function

```

Definition A.26 (Leader Check). For a block b in round r , it b passes the leader check on shard k_i if:

1. In the next round, there exist no leaders (odd round).
2. if there exists a steady leader and there exists sufficient steady votes, if it is in-charge of k_i , it must have a pointer

to b .

3. If there exists sufficient fallback votes, b_i^{r+1} must have a pointer to b .

Or a leader in $r+1$ is already committed, while b is not. Otherwise, the block fails the leader check on shard k_i .

Note that b does not need to be in-charge of shard k_i in this definition.

We now prove using the following propositions that if the leader check is satisfied for a block b , then we have sufficient conditions to be certain that the block in-charge of k_i in round $r+1$ cannot execute before b .

Proposition A.4. Let b be an uncommitted block in round r that persists in round $r+1$. If b passes the leader check for shard k_i , then b will always be committed before b_i^{r+1} .

Proof. Suppose otherwise, b_i^{r+1} executes before b .

Consider the case that b_i^{r+1} is committed due to a committed leader b' in round $r+2$ or after. As b persists in round $r+1$, b' must have a path to b and so it will also commit b . Recall from Definition A.10 that blocks are ordered in non-descending order of the rounds they are created. Therefore, b must be ordered to execute before b_i^{r+1} in $H_{b'}$ and this case is impossible.

Therefore, the committed leader b' must exist in round $r+1$. Since b_i^{r+1} is committed by a leader block in round $r+1$, this implies $b_i^{r+1} = b'$. As b passes the leader check for shard k , the second and third items of the leader check condition guarantee that b' must have a pointer to b . However, this means b will be committed by b_i^{r+1} and by the ordering of blocks in the causal history, b will execute before b_i^{r+1} . \square

This shows that the only block from $r+1$ that can possibly be committed before a block from r that persists in $r+1$ is a leader block in $r+1$. However, if this leader has pointers to the persisting block, it will *not* be committed before the persisting block due to the ordering defined in Definition A.10.

Proposition A.5. Let b be an uncommitted block in round r that persists in round $r+1$. If there exists a leader b' in round $r+1$ where $b \notin H_{b'}$, and b' is already known to be committed, then any uncommitted block in $r+1$ may not be executed before b .

Proof. As b is not in $H_{b'}$, the next elected leader that can commit b must be in round $r+2$ or after. Since b persists in $r+1$, this block must contain b in its causal history and so commit b . Due to the ordering defined in Definition A.10, no blocks in round $r+1$ may be ordered and execute before b even if they exist and are in the next leader's causal history. \square

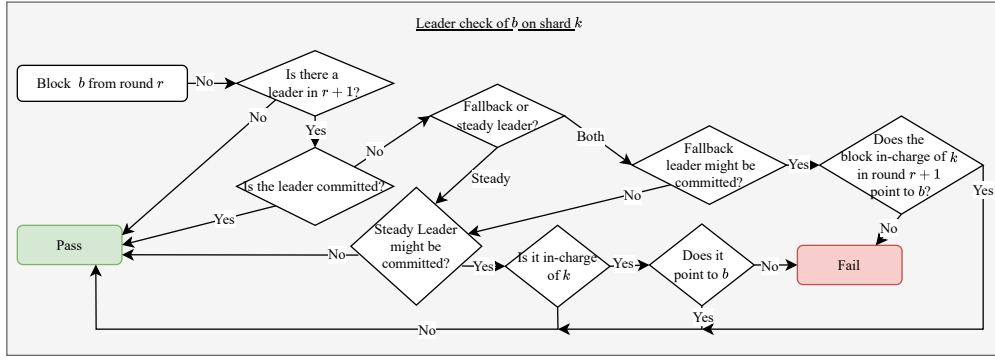


Figure A-1: This Figure illustrates the checks to ensure that if a leader block in-charge of a shard k_i in the immediate subsequent round exists, it must be executed after block b .

A.2.1 STO for Type α

Let us define a notion that will be useful when showing that we can correctly determine STO for a transaction before the parent block is committed.

Definition A.27 (Complete Shard History). *For an uncommitted block b in round r and a shard k_i , let b_i^r be the earliest uncommitted block in charge of shard k_i from round \hat{r} where $\hat{r} \leq r$.*

We say b has Complete Shard History for k_i if and only if b has a path to \hat{b} which includes all blocks from round $r - 1$ to $\hat{r} + 1$ that are in-charge of k_i . We denote this ordered path of blocks as $C_b(i) = [b_i^r, b_i^{r+1}, \dots, b_i^{\hat{r}+1}, b]$, where $|C_b(i)| = r - \hat{r} + 1$.

Observe that by requiring a path containing all the blocks in-charge of k_i , each block on this path also has Complete Shard History. Note that b does not have to be in-charge of k_i to have Complete Shard History for k_i .

Now, we show why Complete Shard History is useful in helping correctly determine STO. Suppose some block b in round r has Complete Shard History for shard k_i . Then b knows of all of the relevant transactions on shard k_i that can be committed. Further, if the next committed leader has a pointer to b , then the leader must also have pointers to all of these blocks, and so these blocks will be committed. Therefore, b knows that all of these blocks must be executed, and so can correctly determine the outcome of executing these blocks before its own execution.

Proposition A.6. *Consider a block b in round r which has complete shard history for k_i . If b passes the leader check on k and persists in $r + 1$, there may not be any block in-charge of k_i which is not in H_b that may be executed before b besides b_i^r .*

Proof. Let b' be the leader that eventually commits b . Observe that as b has complete shard history for k_i , blocks in-charge of k_i but not in H_b must be in round r or after. Now, assume otherwise that some block besides b_i^r in-charge of k_i from some round before r is executed before b . This implies that it must be committed by some leader b'' which commits

before b' . If it is committed by b' , our ordering (see Definition 4.1) guarantees that it is executed after b .

Suppose b'' is in round $r + 2$ or after. Since b persists in round $r + 1$, it must hold that $b \in H_{b''}$ and so is committed by b'' . This directly contradicts that b' is the leader that eventually commits b .

Now, suppose that b'' is in round $r + 1$. Since b passes the leader-check, b'' must have a pointer to b and therefore commits b and again directly contradicts that b is committed by b' .

Finally, if b'' is in round r , the only block in-charge of k that it can commit but is not in H_b is b_i^r (that is, b'' is b_i^r). It cannot possibly commit any block in rounds $r + 1$ or after. \square

Note that b does not need to be in-charge of k_i to have complete shard history for shard k_i .

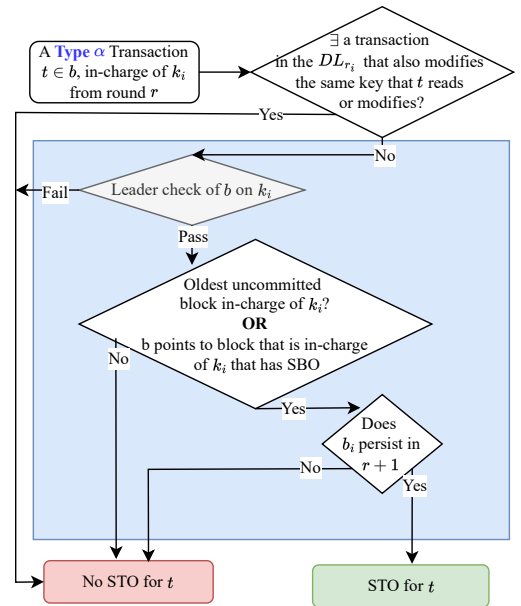


Figure A-2: This Figure illustrates all the sufficient conditions for a Type α transaction to have STO.

Lemma A.2. For a shard k_i , a Type α transaction $t \in b_i^r$ can be determined to have STO in round $r+1$ if all of the following hold:

- There does not exist any transaction in DL_r that modifies k_i .
- b_i^{r-1} has SBO and b_i^r has a pointer to it, or b_i^r is the earliest uncommitted block in-charge of k_i .
- b_i^r passes the leader check on k_i .
- b_i^r persists in round $r+1$.

Proof. Recall that a transaction has STO if its TO matches the execution prefix of that transaction with respect to the leader that eventually commits it. Let b' be the leader that eventually commits b_i^r .

First consider the case where b_i^r is the earliest block in-charge of k_i which is not committed. As b_i^r passes the leader check, by Proposition A.4 it holds that b_i^{r+1} cannot execute before it. Next, as b_i^r persists in round $r+1$, any block in-charge of k_i in round $r+2$ or after cannot execute before it, as any leader committing such a block necessarily has a path to b_i^r and so commits it. By the ordering on the blocks, b_i^r must execute before any block in-charge of k_i in rounds after r . Therefore, it is safe to conclude that b_i^r will execute before any other blocks in-charge of k_i in rounds after r . Therefore, the transaction outcome of t must be equivalent against the execution prefix $b' \langle b(t_i) \rangle$.

Suppose b_i^r is not the earliest uncommitted block in-charge of k_i , but it has a pointer to b_i^{r-1} which has SBO. This implies either b_i^{r-1} is the earliest uncommitted block in-charge of shard k and the earlier argument holds for all its transactions; or it has a pointer to the block that is in-charge of k_i in the preceding round, which has SBO. By applying this argument recursively, this implies that b_i^r has complete shard history of k_i . As b_i^r passes the leader check, by Proposition A.4 and Proposition A.6, we can be sure that there does not exist any block not in H_b in-charge of k_i that may be executed before b_i^r .

Since b_i^{r-1} has SBO, its block outcome is equivalent to its execution prefix with respect to the leader block that commits it. As it must also have complete shard history for k_i (at round $r-1$), its block outcome must execute all of the blocks in-charge of k_i in rounds prior to $r-1$ in order of their rounds. Therefore this also holds true for the execution order of blocks in-charge of k_i by committed leader blocks. So it is clear that when committed by b' , b_i^r must be the next block in-charge of k_i to be executed after b_i^{r-1} . This naturally implies that the transaction outcome of any Type α transaction in b_i^r must be equivalent to the execution prefix of it against b' .

Finally, it is straightforward to observe that these conditions can be determined in round $r+1$.

□

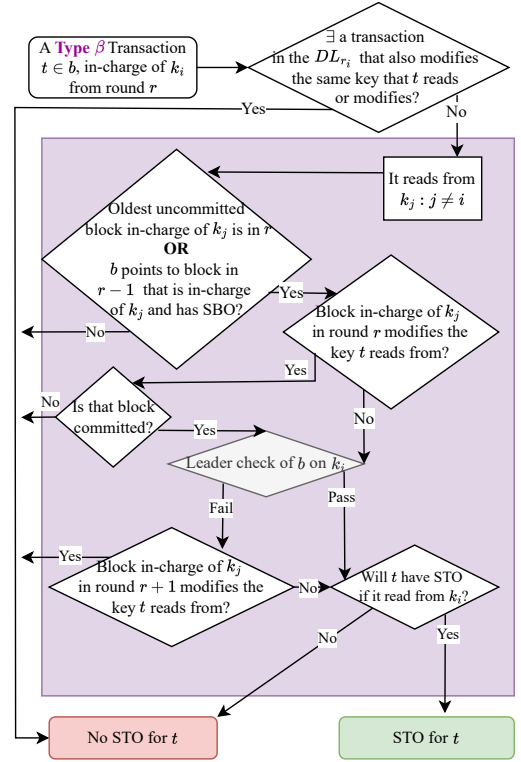


Figure A-3: This Figure illustrates all the sufficient conditions for a Type β transaction to have STO.

Proposition A.3 shows that a certain number of blocks **must** persist in each round. Therefore, it is possible for certain blocks to gain SBO even in the presence of byzantine adversaries and network asynchrony.

A.2.2 STO for Type β

Lemma A.3. Take any two distinct shards k_i, k_j . Let b_i^r be a block in round r in-charge of shard k_i , and let $b_j^{r-1}, b_j^r, b_j^{r+1}$ be blocks in-charge of k_j in rounds $r-1, r, r+1$ respectively. A Type β transaction $t \in b_i^r$ that reads from $k_j^a \in k_j$ can be determined to have STO in $r+1$ if all of the following hold:

1. t meets all of the conditions for STO for a Type α transaction (see Lemma A.2).
2. b_i^r has a pointer to b_j^{r-1} and it has SBO, or b_i^r is the oldest uncommitted block in-charge of k_j .
3. b_j^r does not have a transaction that modifies k_j^a , or b_j^r is known to already be committed by some leader block.
4. b_i^r passes the leader check for k_j or the b_j^{r+1} does not have a transaction that modifies k_j^a .

Proof. Let b' be the leader block that eventually commits b_i^r . As the transaction writes to k_i , we require that t meets the conditions for STO as if it is a Type α transaction, including that it persists in round $r+1$.

Now, since t reads from k_j , we now additionally need to ensure that our read value based on the block outcome of b_i^r will be the same as that for t when it is committed, so that the

transaction outcome of t is equivalent to the execution prefix of t in b' .

First, suppose that either b_j^r does not have a transaction that modifies k_j^a . Given this, executing b_j^r before b_i^r leads to an identical execution prefix of t with respect to b' as if it were executed after. Alternatively, if this does not hold, then we know that b_j^r is already committed by some previous leader. If so, then clearly b_j^r cannot be in $H_{b'}$ (by definition of causal history of a block) and therefore it will not affect the execution prefix of t with respect to b' . In either case, we can thus safely ignore b_j^r as it has no effect on the execution prefix of t with respect to b' .

We have that either b_i^r has a pointer to b_j^{r-1} and b_j^{r-1} has SBO, or that b_j^r is the oldest uncommitted block in-charge of k_j . We then apply a similar argument as Lemma A.2. This implies that b has complete shard history of k_j , and further that the blocks in-charge of k_j in $H_{b_j^{r-1}}$ must be executed in order of their rounds, followed by b_j^{r-1} . Therefore, our read value of k_j^a in the transaction outcome of t will be equivalent to the execution prefix of t with respect to b' , as long as no block in-charge of k_j after round r affects the execution prefix of t with respect to b' .

If b_i^r passes the leader check on k_j , by Proposition A.6, there cannot exist any block in-charge of k_j that can be ordered before b_i^r in the next committed leader's causal history besides b_j^r . Otherwise, if b_i^r does not pass the leader check on k_j then we have that b_j^{r+1} does not have a transaction that modifies t . Then it is clear that executing b_j^{r+1} before b_i^r leads to an equivalent execution prefix of t with respect to b' as if it were executed after. Therefore, b_j^{r+1} will not affect the execution prefix of t with respect to b' . As b' persists in round $r+1$, then no blocks in-charge of k_j will execute before b_i^r .

Therefore, we can conclude that the transaction order of t is equivalent to the execution prefix of t with respect to b' .

These conditions can be evaluated in round $r+1$. \square

A.2.3 STO for Type γ

The following proposition shows how we may infer that two blocks may exist in the causal history of a single leader block before even witnessing the said leader block.

Proposition A.7. *Consider a block b in round r and a block \hat{b} in round \hat{r} , and without loss of generality let $r < \hat{r}$. If both blocks persist in round $\hat{r}+1$ and neither block is in the causal history of any leader block before and up to round $\hat{r}+1$, then both blocks must exist in the same committed leader block's causal history.*

Proof. Since neither block exists in the causal history of leader blocks from before and up to round $\hat{r}+1$, the next possible leader block can contain both must exist after $\hat{r}+1$. Since both blocks persist in round $\hat{r}+1$, that leader block

must have a path to both blocks. Therefore, they must exist in the same committed leader block's causal history. \square

The following proposition shows that we will not incorrectly determine a transaction to have STO when there is uncertainty due to a conflicting transaction in the delay list.

Proposition A.8. *Consider two blocks b_i^{r-1}, b_i^r in-charge of shard k_i in rounds $r-1, r$ respectively.*

If b points to b_i^{r-1} , which has SBO, or if we know b_i^r is the oldest uncommitted block in-charge of k_i , then it implies that DL_r contains all possible transactions that modified keys in k_i .

Proof. If b_i^r was the oldest uncommitted block in-charge of k_i and it is in r , then any other possible block in-charge of k_i prior to r must have been committed and therefore known.

Similarly, if b_i^{r-1} has SBO, it implies that b has complete shard history of k_i . Therefore, there does not exist an uncommitted block in-charge of k_i from before round r that is not in H_b . Therefore, all blocks in-charge of k_i prior to r is known as well.

Since all blocks that are in-charge of k_i is known, there must not exist a transaction that would be in DL_r that we do not know. \square

Definition A.28 (Type γ sub-transactions execution order). *Consider a Type γ transaction comprised of two sub-transactions t_1, t_2 that exist in blocks b_1, b_2 from rounds r_1, r_2 respectively.*

- If $r_1 = r_2$, and b_1, b_2 are in the same leader's causal history, then t_1, t_2 are executed in some deterministic order between the 2 transactions.
- If $r_1 < r_2$, and b_1, b_2 are in the same leader's causal history. Then t_1 will be executed concurrently with t_2 . That is to say, it is not executed along with the other transactions in b_1 .
- If b_1, b_2 are committed by different leaders. The one committed earlier executes at $\max(r_1, r_2)$ with the one committed later.

Lemma A.4. *Consider a Type γ transaction of 2 sub-transactions $t = [t_1, t_2]$ where $t_1 \in b_1, t_2 \in b_2$ in round r . The transaction t can be determined to have STO in $r_1 + 1$ if all of the following hold:*

- Both t_1 and t_2 may be evaluated to have STO independently.
- Every other transaction in b_1 and b_2 have STO.
- The conditions of Proposition A.7 is met and so there exists some eventually committed leader block b' such that both t_1, t_2 exist in $H_{b'}$.

Proof. First, given that the conditions of Proposition A.7 hold, we now assume b_1 and b_2 are committed by the some leader block b' , and without loss of generality, let the prime transaction be t_2 . Given that blocks of the same round are ordered deterministically, it is known that all blocks will order b_1 before b_2 and this assumption can be safely made.

Now, given that both t_1 and t_2 satisfy the requirements to have STO independently, the transaction outcome of t_1 and t_2 are each equivalent to their respective execution prefix with respect to b' . However, now t_1 will be concurrently executed with t_2 instead of following the typical ordering on the blocks and transactions.

Given that all other transactions in b_1 and the transactions prior to t_2 in b_2 will be guaranteed to execute *before* t_1 , their execution may now affect the transaction outcome of t_1 . Given that all of these transactions have STO, their transaction outcome is equivalent to their execution prefix with respect to b' and therefore, it can be correctly determined their effect on transaction outcome of t_1 . Therefore transaction outcome of t_1 now, when ordered concurrently with t_2 is indeed equivalent to its execution prefix with respect to b' . This is illustrated with Fig. 8.

All of the above may be evaluated in round $r + 1$. \square

Lemma A.5. Consider a Type γ transaction of 2 sub-transactions $t = [t_1, t_2]$ where $t_1 \in b_i^{r_1}$ and $t_2 \in b_j^{r_2}$, and without loss of generality, $r_1 < r_2$.

$t = [t_1, t_2]$ can be determined to have STO in round $r_2 + 1$ if all of the following hold:

- t_1 independently has STO when considering it to be in $b_j^{r_2}$. This condition includes that $b_i^{r_2}$ must be observed.
- t_2 independently has STO.
- Every other transaction in $b_j^{r_2}$ has STO.
- Furthermore, all other transactions in $b_i^{r_2}$ have STO given that t_1 were to be in $b_i^{r_2}$ instead of $b_i^{r_1}$.
- The conditions of Proposition A.7 is met and so there exists some eventually committed leader block b' such that both t_1, t_2 exist in $H_{b'}$.

Proof. First, given that the conditions of Proposition A.7 hold, we now assume that $b_i^{r_1}$ and $b_j^{r_2}$ are committed by the same leader block b' .

Since $r_1 < r_2$, t_1 will be moved to execute concurrently with t_2 instead of in $b_i^{r_1}$. Given that $b_i^{r_2}$ is observed, then now let t be removed from $b_i^{r_1}$ and placed as the last transaction in $b_j^{r_2}$ instead. Now, this becomes an instance which is covered by Lemma A.4. Given that the conditions of this lemma are met, then it holds that the conditions of Lemma A.4 are satisfied as well. Therefore, the transaction order of t_1 will be equivalent to its execution prefix with respect to b' .

There is one technical detail that is different, since it is known that t_1 will now be executed with t_2 (as if in $b_j^{r_2}$). Therefore, its presence in the delay list can now be safely ignored when considering STO for transactions in blocks in-charge of k_i in between and including rounds r_1 and r_2 , as it is guaranteed that this transaction will only execute after these blocks.

If the above conditions are true, STO may be evaluated at round $r + 1$. \square

Lemma A.6. Type α, β, γ transactions may be evaluated to have STO before the leader b' that includes them in its causal history is committed.

Proof. For a Type α, β transaction in round $r + 1$, it may be evaluated to have STO in round $r + 1$ as per Lemmas A.2 and A.3. Since the leader checks were passed, the earliest the transaction may be committed is round $r + 2$, where b' is a steady leader.

For Type γ transactions, that exist in two parts b_1, b_2 from rounds r_1, r_2 respectively. STO may be evaluated at round $\max(r_1, r_2) + 1$. whilst the leader may exist in the earliest round $\max(r_1, r_2) + 1 + 1$ and committed in round $\max(r_1, r_2) + 1 + 2$. \square

Therefore, by Lemma A.6, early finality is possible for Type α, β, γ transactions.

B Extending Type β/γ transactions

Extending Type β transactions to read from a set of shards is quite simple. Suppose we have a transaction $t \in b_i^r$ that reads from a set of shards $\mathcal{K} = \{\dots\}$. $\forall k_j \in \mathcal{K}$, b_i^r needs points to a block b_j^{r-1} that has SBO, and it passes the leader check on shard k_j . If these conditions are achieved, it's clear to see how Lemma A.3 may be extended to prove that t has STO.

Extending Type γ transactions to be an n -tuple across n shards is quite simple as well. Suppose all n sub-transactions exist in the same round, have SBOs, and we can ensure that all sub-transactions will eventually be in the causal history of the same committed leader. Per Lemma A.4, the Type γ transaction has STO as well. A similar argument may be applied to Lemma A.5 when all n sub-transactions exist in different rounds, but is eventually committed by the same leader.

C Future Work: A Finer Grained Lemonshark

For clarity, Lemonshark presents *sufficient* conditions for determining whether a block has SBO, evaluated recursively starting from the earliest uncommitted block in each shard. Intuitively, SBO is “inherited” from one block to the next: a block cannot have SBO if the uncommitted block in-charge of the same shard in the previous round does not.

However, this inheritance requirement is stronger than necessary for individual transactions. Consider blocks b_1, b_2 —the

first and second uncommitted blocks in charge of a shard—and a type α transaction $t \in b_2$ that modifies a key untouched by any transaction in b_1 . Transaction t can achieve STO as long as b_2 persists and passes the leader check, regardless of whether b_1 has SBO or whether b_2 references it.

This observation reveals that Lemonshark’s current conditions are sufficient but not necessary. Deriving tight *necessary* conditions for STO evaluation remains an important direction for future work.

D Missing blocks, Weak links, Orphaned and Dangling Blocks

Determining the “*oldest uncommitted block*” for a shard in the presence of Byzantine nodes requires identifying whether missing blocks are genuinely absent or exist without a node’s knowledge.

Missing blocks can be ascertained by a node querying the remaining nodes to determine whether they voted in the second phase (vote phase) of the reliable broadcast [17]. If fewer than $2f + 1$ nodes voted ($< f + 1$ out of the $2f + 1$ responses obtained), such a block will never exist and can be categorized as missing.

However, if at least $2f + 1$ ($\geq f + 1$ out of the $2f + 1$ responses obtained) nodes voted, that block might exist. This is not problematic if all blocks in subsequent rounds are known and none reference the missing block. We refer to those as orphaned blocks.

Dangling blocks (blocks pointed to by only a small subset of blocks and never committed) are categorized similarly. Bullshark [27] allows blocks to reference blocks from non-immediate previous rounds via pointers referred to as “weak links.”

These weak links do not participate in consensus; instead, they are used to help orphaned or dangling blocks eventually get committed. However, since blocks may be referenced via weak links almost arbitrarily, we disallow such links in Lemonshark, as they enable arbitrary inclusion of blocks into a node’s causal history.

The problem with dangling blocks is that if they are not committed, they will always be the *oldest uncommitted block* in-charge of a particular shard, preventing other blocks in-charge of the same shard from ever gaining SBO.

In this section, we will define *limited look-back* as a potential solution to handle such cases, effectively acting as a high-water mark that eventually excludes these dangling blocks from consideration.

We first define Sorted Causal History with Limited Look-back to be used in-place of Sorted Causal History.

Definition D.1 (Sorted Causal History with Limited look-back (v)). *Suppose the last known committed leader is in round r' , where the next possibly committed leader is in round $r' + 2$. Consider a block b in round $r > r'$ and let v be a publicly known constant. b ’s Sorted Causal History with Limited look-back LH_b is defined to be all blocks in H_b that are from round*

$r' + 2 - v$ or after. We denote $r' + 2 - v$ as the watermark m_b for b .

Consider some block b in round r and the leader block b' in round r' that eventually commits b . One issue that can potentially occur with the above definition is that a node’s local view of LH_b may include blocks that are not in $LH_{b'}$, if they have different watermarks. Therefore, we need to show that for any block $b \in LH_{b'}$, their Sorted Causal History with limited look-back excludes blocks from the same rounds, that is, their watermarks are the same.

Lemma D.1. *Consider a leader block b' from round r' that eventually commits. For all $b \in LH_{b'}$, the watermark m_b of block b is identical to watermark $m_{b'}$ of block b' .*

Proof. Suppose that for any node’s local view of the DAG, there exists a block $b \in LH_{b'}$ where its watermark differs from $m_{b'}$. By the definition of watermark of b and b' , this implies that there exist two distinct next possibly committed leaders from 2 different rounds, which is absurd. \square

The next question to address is whether for a given block b (from round r), the Sorted Causal History with Limited Look-back of b will differ for two distinct nodes. Observe that the only factor that changes the local view of the watermark is the knowledge of the latest committed leader. If a node evaluates a block to have met the criteria for SBO, the criterion will hold regardless of which leader prior to r is newly committed. This consistency is illustrated in Appendix A.2.

In other words, suppose the view of the watermark is for a fixed block b for a node is m_{b1} and it evaluates b to have SBO. Even if the view of the watermark for b is m_{b2} where $m_{b2} > m_{b1}$ for another node, it will not alter the execution prefix of b with respect to the leader that eventually commits it. Intuitively, this holds because the node with the lower watermark must account for a larger set of conflicts.

It is also important to note that even in the case outlined using Definition 4.1, it is not possible for a node to initially evaluate a block as meeting the SBO criterion, and later, after learning of a more recently committed leader, revise this evaluation to not meet the SBO criterion. This holds as well when utilizing Definition D.1.

Thus, the use of Sorted Causal Histories with Limited Look-back eventually eliminates dangling blocks from consideration. By constantly updating the threshold for the *oldest uncommitted block*, this mechanism refreshes the possibility for blocks to meet the SBO criterion.

E Experimental Discussion

E.1 Rationale for Randomizing Faulty Nodes

There are a few reasons to randomize which nodes are faulty; firstly, in the case where there is a single fault: In the original Bullshark [27] implementation, the steady leader is elected in a round robin manner. Which means it is possible that the supposed faulty node will never be the 2nd steady

leader of a wave, causing the system to never enter fallback-mode. Secondly, the original implementation orders the node randomly and then selects the leader in a round-robin manner, enabling the previous case even when $f > 1$. Artificially improving the performance of the protocol when faults exist.

Lastly, since Lemonshark requires a recursive chain of SBO for a block to gain SBO (if it was not the latest un-committed block). Randomizing the failures will result in a worse but fairer representation of Lemonshark.

E.2 Major Code Differences

Here we discuss some changes made to Bullshark’s Code.

1. We were unable to get the claimed performance when utilizing parameters as described in their paper (utilizing their code). However, we achieved similar results when we utilized the parameters present in their repository⁵. We believe this could either be a mistake on their part or one of the various bugs in their fallback version of the protocol.
2. Similar to Narwhal-Tusk [22], nodes perform 1 round of one-to-all broadcast to create batches, and another one-to-all broadcast to form blocks including those batches. Each batch contains up to 500kB of transactions, but can be represented by a 32B hash. In their code, whenever producing a block, they flushed their whole queue, resulting in blocks containing an unbounded number of batches (and an extremely large block including many hashes). We set an upper-limit (bounded by the block size); therefore, we exhibit a queuing behavior in our results when the client transaction rate is increased.
3. Instead of picking the last f nodes to be faulty nodes, we randomized the selection. Furthermore, we also randomized the steady leader election as compared to a simple round-robin rotation (per Bullshark). However, we add a restriction that no two consecutive steady leaders are the same. This is to introduce a more normalized yet realistic failure behavior. Previously, because the rotation of leaders was static, it was possible for the system to never enter fall-back mode in some cases, skewing the results. Our procedure allows the state to enter *fall-back* mode reliably when $f = 1$, making it more aligned with an adaptive adversary.

E.3 Additional Experiment results

Varying Cross-shard probability in Lemonshark

In §8.2 we utilized Cross-shard probability= 50%, denoting the percentage of Type β transactions; in Fig. A-4 we show the effects of varying that rate. Even at 100%, we enjoy $\sim 13\%$ E2E latency reduction and $\sim 18\%$ consensus latency reduction.

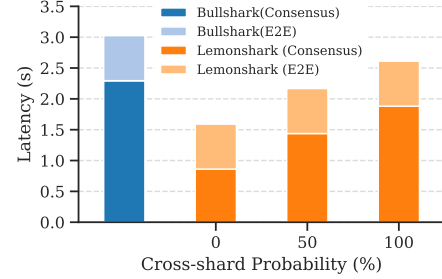


Figure A-4: Performance of Lemonshark with moderate amount of Type β transactions, (Cross-shard count = 4, Cross-Shard Failure = 33%) while varying Cross-shard probability.

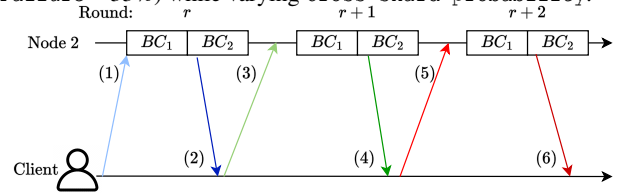


Figure A-5: For illustration, we break the RBC of our block into two one/all-to-all broadcast instances BC_1, BC_2 ; akin to the 2-phase nature of Bracha’s reliable broadcast [17]. The client first sends in a transaction t_1 . (2) After the first BC_1 , the process provides a speculative outcome for t_1 in the form of to_1 . (3) The client then sends in the next transaction t_2 with the requirement that it only executes if the execution of t_1 yields to_1 . (4,5,6) similar steps repeat for t_2, t_3 .

E.4 Cost of Experiments

As illustrated in the appendix of Narwhal-Tusk [22], running these experiments can very easily get costly quickly.

F Pipelining Dependent Client Transactions

Beyond the scope of our main insight in §5, we observe that if a client has a chain of transactions t_1, \dots, t_l such that every t_i requires the outcome of the previous transaction t_{i-1} , a client will typically need to send t_1 , wait for its finalized outcome when committed, before sending t_2 , and so on. Therefore, the total latency incurred is at least the time required for l distinct leaders to be committed.

Since a RBC primitive may be decomposed into two phases [17] of one-to-all broadcast. Suppose the process receiving the transaction provides the (unfinalized and potentially unsafe) speculative TO for t_1 in b_1 for round r_1 during the first phase. In that case, the client may send a tentative transaction t'_2 that executes conditionally on the speculated outcome of t_1 being the one specified in t'_2 .

Before the first phase of the *next* block $b_2 : t'_2 \in b_2, b_2 \in H_{b_1}$ in round $r_1 + 1$ can be proposed, a similar operation is performed for $t'_3 \in b_3$, and so forth.

Such pipelined behavior can potentially achieve much lower latencies for the entire chain of transactions t_1, \dots, t_l ; we illustrate this in Fig. A-5. It is important to note that due to our sorting algorithm in Definition 4.1, $\forall b : b_1, b_2 \in H_b, b_2$ is always ordered after b_1 in H_b .

Eventually, when the block is committed or has satisfied

⁵ <https://github.com/asonnino/narwhal/tree/bullshark-fallback/benchmark/data/latest>

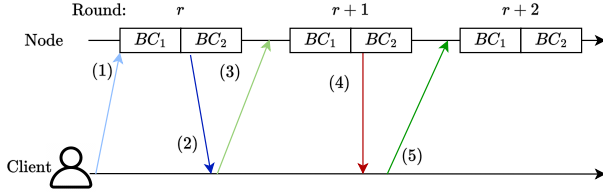


Figure A-6: Catching the next bus: (1) Our client sends a Type β transaction to the process. (2) After the first BC_1 , the process sends a speculative outcome. (3) The client uses this speculative outcome to send its 2nd causally related transaction. (4) At $r+1$ we learn that our read will be stale, and the process informs the client (5) The client then re-sends the 2nd transaction in the immediate subsequent block, incurring a single block worth of delay compared to an entire consensus latency worth.

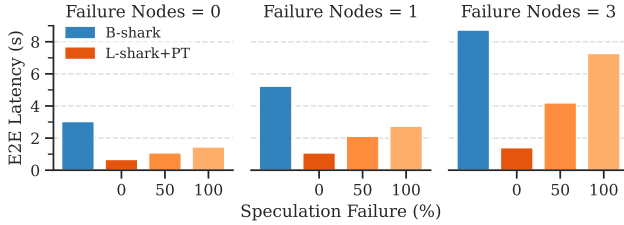


Figure A-7: Performance of Pipelined client transactions (L-shark + PT) when varying the number of crash-faults.

the conditions for early finality, the finalized outcome of t_1 is communicated to the client, whereupon two cases may occur:

1. **The finalized outcome matches the speculated outcome provided previously:** In this case, the client simply continues, knowing that t'_2 has been submitted and will be executed in due time.
2. **The finalized outcome does *not* match the previously speculated outcome:** In this case, t'_2 will only execute if the speculative outcome of t_1 aligns with the one included in t'_2 . If not, t'_2 is aborted, as are any subsequent transactions that depend on its speculated outcome. The client then immediately submits a new transaction, t''_2 , based on the finalized outcome of t_1 , thereby restarting the chain of transactions.

Because cascading failure is intrinsic to the transaction chain itself, the above conclusions can be reached independently by each process without requiring coordination. Even if an unexpected outcome occurs, the latency remains upper-bounded by the rate of normal consensus progress via leader election. Thus, in the worst case, this pipelined behavior incurs no additional penalties.

F.1 Pipelining and Lemonshark

Lemonshark allows processes to determine whether transactions may have STO or whether they definitively cannot have STO. In such cases, the provided speculative TO may be known to be definitively inaccurate before commitment. This can be communicated to the client before finality, thereby al-

lowing the client to resubmit its chain of transactions earlier; we illustrate this in Fig. A-6.

However, transactions that may execute based on some condition are analogous to Type γ transactions, in that the conditional execution of a transaction may affect other transactions. As such, all contingent transactions are added to the Delay List as well. They are removed when they are finalized or removed entirely when cascading failure occurs.

F.2 Evaluation

We also evaluated how pipelined dependent transactions may be fixed by fixing a moderate amount of Type β transactions (Cross-shard count = 4, Cross Shard Failure = 33%), and varying the number of failures. We also added a parameter Speculation Failure to denote the probability that the speculated outcome deviates from the finalized outcome if STO is not possible. In the best case, we saw an $\sim 80\%$ improvement in latencies when there are no faults, but even in the worst case we still saw $\sim 11\%$ lower E2E latencies. This is because of the phenomenon we describe in Fig. A-6.