Masking Ordering Failures in BFT SMR via Proactive Pre-Commit Execution

Jianting Zhang
Purdue University

Alberto Sonnino
Mysten Labs/UCL

Lefteris Kokoris-Kogias
Mysten Labs

Aniket Kate
Purdue University/Supra Research

Abstract

Modern Byzantine fault-tolerant state machine replication (BFT SMR) systems adopt a decoupled BFT consensus process to separate data dissemination from transaction ordering as it enables efficient (asynchronous) dissemination even when ordering fails intermittently under partial synchrony. Nevertheless, they may still suffer from high transaction confirmation latency as the transaction-execution process waits for the ordering process to complete: the execution process does not proceed even when transactions are disseminated if the ordering process stalls.

We propose Pufferfish, the first BFT SMR system that effectively masks intermittent ordering failures in practice. Pufferfish introduces a *pre-commit* execution scheme that enables replicas to speculatively execute transactions even during the ordering process stalls. These pre-commit execution results can be directly committed, if correct, when the ordering failures are resolved. To achieve this, Pufferfish builds an adaptive probabilistic speculation mechanism on top of a DAG-based BFT consensus protocol, enabling replicas to predict and speculatively execute transactions ahead of confirmed ordering. Additionally, Pufferfish adopts a commit-aware snapshot mechanism to minimize the overhead of transaction re-execution in cases of speculation failures. To demonstrate the effectiveness of Pufferfish, we implement and evaluate it on a geo-distributed AWS environment. The evaluation results show that Pufferfish achieves faster recovery and 1.36x speedup on the p99 transaction confirmation latency compared to the state-of-the-art BFT SMR in the presence of ordering failures. Even under normal execution, Pufferfish can achieve a 1.58x speedup on transaction confirmation latency under a transaction workload of 80k tps.

ACM Reference Format:
Jianting Zhang, Alberto Sonnino, Lefteris Kokoris-Kogias, and Aniket Kate. 2026. Masking Ordering Failures in BFT SMR via Proactive Pre-Commit Execution. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With significant implications in practice, blockchain has been extensively studied in the distributed systems literature in the last decade. At the core of blockchain is Byzantine Fault-Tolerant State Machine Replication (BFT SMR), which allows a group of replicas to consistently and continuously handle transactions from blockchain users. In the BFT SMR problem, replicas run a BFT consensus protocol to disseminate and order transactions. With the consistent transaction order established by consensus, correct replicas can execute transactions and replicate the states consistently.

Modern blockchains/BFT SMRs are primarily designed to operate in the partial synchrony network model, in which messages are guaranteed to be delivered within a known bounded delay after an unknown Global Stabilization Time (GST) [23]. Theoretically, these partially synchronous BFT SMRs are expected to reach agreements on transaction orders *forever* after GST, under benign conditions (e.g., the leader is correct). However, in practice, this happy regime only holds *intermittently*—numerous attacks can cause ordering failures [2, 8, 18, 27, 29, 39, 59]. It is observed that the network can frequently switch between synchrony and asynchrony, leading to intermittent failures on transaction ordering [28].

To adapt to these intermittent ordering failures, modern blockchains employ *decoupled* BFT consensus protocols, including multi-chain BFT [22, 28] and a family of Directed Acyclic Graph (DAG)-based BFT [5, 6, 20, 36, 57, 58, 60, 61]. The decoupled BFT consensus protocols separate data dissemination from transaction ordering. In particular, during the asynchrony periods, the transaction ordering may halt, but replicas can still keep disseminating transactions without waiting for the completion of ordering [20]. After the network recovers and becomes synchronous, replicas can establish an order on the disseminated transactions via extremely lightweight messages, such as signed-hash digests [25, 66, 78] and block references [5, 6, 28, 57, 61].

In spite of the numerous advancements in consensus protocols to handle intermittent ordering failures, there remains a critical gap in efficiently executing transactions during these failures. For instance, Sui [64], a production blockchain employing a DAG-based consensus protocol, adopts an execution-after-consensus scheme, where transactions are executed only after the transaction order is finalized. In this paper, we find that the intermittent ordering failures could significantly affect transaction latency in the existing blockchains. This is not surprising, as replicas might need to execute an extensive backlog of transactions that were accumulated by data

dissemination during the asynchrony periods once the network recovers. To elaborate, unlike the data dissemination that does not rely on transaction ordering to proceed, transaction execution must be aligned with the ordering results to ensure the state consistency among correct replicas. Before the transaction order is established, those ever-growing disseminated transactions can stress the execution overhead once the network recovers, leading to a severe degradation of the transaction confirmation latency.

This paper aims at filling this gap. We propose Pufferfish, a novel BFT SMR system that effectively reduces the transaction confirmation latency under intermittent ordering failures while maintaining high transaction throughput. The core idea of Pufferfish is to incorporate a *pre-commit* execution scheme into a decoupled BFT consensus, allowing replicas to speculatively execute transactions even during ordering stalls. By the time the ordering stalls are resolved, Pufferfish would have already performed the bulk of the computation, allowing near-instantaneous commitment. However, designing an effective pre-commit execution scheme for a BFT SMR system is non-trivial. Pufferfish must ensure that as many speculative execution results as possible remain valid; that is, transactions should be speculatively executed in the same order as the final consensus order. Otherwise, those results become invalid, and replicas must re-execute the backlog after ordering recovers, reducing the system to the baseline execution-after-ordering design.

Pufferfish addresses the above challenges via three techniques. First, Pufferfish employs a DAG-based consensus protocol to achieve efficient data dissemination while preserving an *order continuity* property, which ensures that speculative orders remain stable and are likely to be prefixes of the final consensus order (Section 4.1). Second, Pufferfish introduces an *adaptive probabilistic speculation (APS)* mechanism to allow replicas to predict the most likely consensus order, significantly reducing the probability of re-execution caused by the inconsistent speculative order (Section 4.2). Specifically, with the heuristics derived from the disseminated DAG of blocks, replicas can adaptively update the speculative order for later pre-commit execution, making it always most likely consistent with the consensus order. Third, Pufferfish designs a *commit-aware snapshot* mechanism to achieve fast recovery when pre-commit execution fails, without introducing heavy memory or storage overhead (Section 4.3). Specifically, instead of taking snapshots after pre-commit execution for every speculative order, replicas only take a lightweight snapshot when they suspect an incoming speculative order is invalid using a reputation-based decision strategy. This avoids excessive snapshots and reduces memory and storage overhead. Once the speculative order is indeed invalid, replicas can quickly roll back to the latest valid snapshot and re-execute only a smaller set of transactions whose order is invalid, thereby minimizing the overhead caused by mis-speculations.

To demonstrate its effectiveness, we implement Pufferfish on top of Mysticeti [6], a DAG-based BFT consensus protocol deployed in the Sui blockchain [64], and evaluate it against a representative BFT SMR system in a geo-distributed AWS environment. The evaluation results show that Pufferfish can achieve 1.36x-1.58x speedup on transaction confirmation latency compared to the representative BFT SMR system.

We summarize our contributions as follows:

- We present Pufferfish, which, to the best of our knowledge, is the first BFT SMR system designed to effectively circumvent the performance degradation under intermittent ordering failures in practice.
- We design an adaptive probabilistic speculation mechanism to help replicas predict the speculative execution order effectively, and a commit-aware snapshot mechanism to minimize the overhead of transaction re-execution in cases of speculation failures.
- We implement Pufferfish on top of Mysticeti and evaluate it on a geo-distributed AWS environment. The evaluation results show that Pufferfish can achieve 1.58x speedup on transaction confirmation latency under common cases, and faster recovery with 1.36x speedup on the tail p99 transaction confirmation latency in the presence of ordering failures compared to the representative BFT SMR.

2 Background and Motivation

2.1 BFT SMR

BFT SMR is the standard primitive for building fault-tolerant distributed systems, such as blockchains. In this model, clients submit transactions to a set of replicas to induce state transitions (in the blockchain context, the system state typically comprises a key-value store mapping client identities to data such as account balances and smart contract storage). A BFT SMR system must ensure that this state remains consistent across all correct replicas while maintaining the ability to process new client requests. More formally, let $St^i_{\vdash k}$ denote the system state maintained by replica R_i after executing a sequence of transactions up to index k, the BFT SMR must guarantee the following properties:

- *Safety:* If sequences of transactions $(tx_1, \cdots, tx_m)$ and $(tx'_1, \cdots, tx'_{m'})$ are committed by two correct replicas R_i and R_j, then $tx_n = tx'_n$ and $St^i_{\vdash n} = St^j_{\vdash n}$ for all $n \leq min\{m, m'\}$, namely, any two correct replicas commit the same prefix transaction list and states.
- *Liveness:* If a transaction tx is sent to correct replicas, tx will be eventually committed by all correct replicas.

The transaction processing pipeline in BFT SMR can be decomposed into three repeated tasks [17, 78]: (i) *Data dissemination*, a bandwidth-bound task where replicas propagate transactions to other replicas; (ii) *Ordering*, a network-sensitive task where replicas reach consensus on a total order

of the disseminated transactions; (iii) *Execution*, a computation-intensive task where replicas execute transactions in the consensus order to update the system state. A BFT SMR system is typically implemented with a BFT consensus protocol (responsible for data dissemination and ordering) and an execution engine (responsible for execution).

2.2 Partially Synchronous SMR in Practice

Modern BFT SMR systems primarily operate in a partially synchronous network model [23], where message delays are unbounded before an unknown Global Stabilization Time (GST), and bounded by a known delay Δ after GST. Informally, there exists a magic time GST, before which the network is asynchronous and after which it is synchronous.

In realistic deployment, the partially synchronous BFT SMR accommodates this network model by using timeouts: replicas wait for a certain time duration before they conclude that the system makes no progress and take actions, such as performing a view change mechanism. To make sure that all correct replicas can enter the same view to perform the ordering task after GST, the timeout has to be set delicately, which is typically measured and associated with the network delay Δ (e.g., the timeout is set to be 8Δ in [57]). With the delicate setting, the timeout is *expected* to not be triggered after GST, unless in the event of leader failure.

Intermittent Ordering Failure. However, in practice, this happy regime is not held, and timeouts can be triggered *intermittently*. The network delay Δ can fluctuate due to many factors such as network failures (e.g., packet loss) and network adversaries (e.g., DDoS attack) [2, 8, 18, 27–29, 39, 59]. Moreover, malicious replicas can trigger timeouts by not proposing ordering proposals. As a result, the ordering task can fail intermittently whenever the timeout is triggered. In this paper, we refer to such a phenomenon as *intermittent ordering failure*.

2.3 Decoupled BFT Consensus

Decoupled BFT consensus protocols are designed to (partially) mitigate intermittent ordering failures by separating data dissemination from ordering logic. This separation allows replicas to continue disseminating transactions even when ordering fails intermittently. Modern decoupled consensus protocols generally fall into three categories (we defer a more detailed comparison to Appendix A):

Batch-based BFT protocols [25, 32, 35, 42, 66, 74, 78] disseminate transactions in batches, independently of ordering. Replicas continuously group new transactions into batches and broadcast them for data availability. When performing ordering, replicas replace full transactions with lightweight digests of batches in ordering proposals. As transactions can be disseminated asynchronously, this design enables continuous dissemination despite ordering failures.

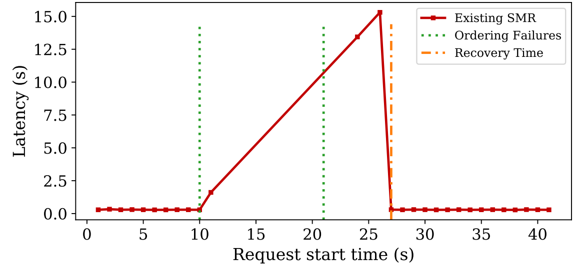


Figure 1. The transaction latency and recovery time of existing SMR systems under intermittent ordering failures.

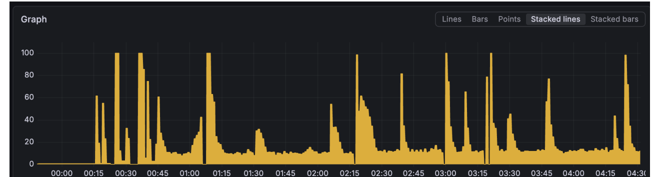


Figure 2. Execution jitter during the Sui testnet incident (June 2025). The x-axis shows time (incident started at 00:15) and the y-axis shows transaction latency overhead (ms).

Multichain BFT protocols [22, 28] use multiple parallel chains for dissemination. Each replica packages its received transactions into blocks appended to its own chain and broadcasts these blocks to certify the availability of both the block and its chain history. Ordering is performed separately by replicas running an ordering/consensus protocol, using as input the signed digest of the latest block from each chain. Compared to batch-based protocols, multichain protocols require a bounded size of data in an ordering proposal since only the latest block digest of each chain is needed.

DAG-based BFT protocols [5, 6, 20, 36, 57, 58, 60, 61] disseminate transactions in a structure of Directed Acyclic Graph (DAG), where blocks of transactions are DAG vertices and the references among them form edges. Each replica creates and disseminates new blocks in *rounds*, with each block referencing a quorum of blocks from the previous round. With the established DAG, replicas perform the ordering task by locally interpreting the DAG, e.g., determining whether a block is sufficiently referenced in subsequent rounds to be committed and ordered. Compared to other decoupled BFT consensus, DAG-based consensus can achieve more efficient ordering due to the elimination of view change [79].

2.4 Motivation

Although decoupled BFT consensus protocols take an important first step toward mitigating intermittent ordering failures, they do not fully resolve the problem. This can still severely impact transaction execution.

Unlike data dissemination, which proceeds independently of ordering, transaction execution relies on the ordered output of the consensus protocol to ensure safety. When ordering fails, replicas cannot obtain new ordered transactions to execute, leaving computational resources idle. More importantly, decoupled BFT consensus protocols can exacerbate

this issue: transactions continue to accumulate during the failure period, and once the network recovers and ordering resumes, replicas must process a large backlog. This burst of deferred execution can overwhelm computational resources, significantly increasing transaction latency. Figure 1 illustrates this effect by showing the transaction confirmation latency and recovery time of existing BFT SMR systems under intermittent ordering failures (evaluation settings are detailed in Section 6.2). During the ordering failure period (10s-21s), transactions accumulate in the system, and after recovery, existing SMR systems require a prolonged catch-up phase (around 6 seconds) to drain the backlog, resulting in a substantial latency increase. Similarly, the Sui testnet experienced a roughly one-day halt in June 2025 after replicas failed to locally confirm transactions and had to process a large backlog (as shown in Figure 2), illustrating how ordering failures can severely disrupt progress even in modern blockchain deployments [63].

To fully address the challenges introduced by intermittent ordering failures, it is crucial to design a BFT SMR system that effectively adapts to such failures in practice. However, upon reviewing related studies, we find a notable absence of comprehensive investigations into this problem. This paper aims to fill this gap by presenting a novel BFT SMR.

3 Model, Challenges, and Overview

This paper presents Pufferfish, a novel BFT SMR system that masks performance degradation caused by intermittent ordering failures in practice. To achieve this, Pufferfish incorporates a *pre-commit execution scheme* into a decoupled BFT consensus protocol, enabling replicas to speculatively execute transactions during data dissemination without waiting for final ordering (We use pre-commit and speculative execution interchangeably in the rest of this paper). By allowing execution to proceed even when ordering stalls, replicas can later directly commit correct pre-commit execution results once ordering completes. This design eliminates the execution stall and significantly reduces the backlog-processing overhead after recovery. In this section, we present the system model, key challenges, and an overview of Pufferfish.

Models. In Pufferfish, we consider a group of n=3f+1 replicas $\mathcal{R} = \{R_1, R_2, \cdots, R_n\}$, of which up to f are *Byzantine*. The Byzantine replicas can behave arbitrarily to compromise the system, but they have bounded computation and cannot break the cryptographic primitives. The remaining replicas are *correct*. We assume a public-key infrastructure with digital signatures for authentication. As in prior partially synchronous BFT SMR systems [10, 62, 78], the system operates under partial synchrony [23], where liveness is guaranteed only during periods of synchrony.

Challenges. Designing an effective pre-commit execution scheme to mitigate ordering failures is non-trivial. The core challenge is to ensure that speculative execution remains largely consistent with the consensus order, so that pre-commit execution results can be directly committed without re-execution. This challenge stems from the inherent dependence of execution on ordering. While data dissemination is order-independent, execution must follow a total order consistent with consensus to ensure safety. If speculative execution deviates from the final order, its results become invalid, forcing replicas to roll back and re-execute transactions. In the worst case, this reduces Pufferfish to the unoptimized design, negating any benefit in masking ordering failures.

Overview. To effectively mask ordering failures, Pufferfish is built on three key components: (i) a dedicated DAG-based consensus protocol; (ii) an adaptive probabilistic speculation (APS) mechanism; (iii) a commit-aware snapshot mechanism.

DAG-based consensus (section 4.1). Pufferfish adopts Mysticeti [6], a state-of-the-art DAG-based BFT consensus protocol. Its decoupled design enables efficient transaction dissemination even when ordering stalls. More importantly, the DAG structure provides an order continuity property: subsequent ordering instances naturally extend prior (even failed) ordering attempts through causal dependencies encoded in the DAG. As a result, new ordering decisions tend to preserve previously implied orders, reducing the likelihood that speculative execution must be discarded. This property is critical because it increases the fraction of speculative execution results that remain valid and can be committed directly.

Adaptive probabilistic speculation (Section 4.2). While order continuity improves stability, it does not guarantee that speculative execution matches the final consensus order. Replicas must therefore predict a likely speculative execution order. To this end, Pufferfish introduces an adaptive probabilistic speculation (APS) mechanism. APS leverages structural information in the DAG (e.g., implicit voting patterns over leader blocks) to estimate the likelihood of different ordering outcomes. Based on the likelihood estimation, each replica adaptively selects a speculative execution order that is most likely consistent with the final consensus order, thereby maximizing the valid pre-commit execution results.

Commit-aware snapshot (Section 4.3). Despite careful speculation, mis-speculation is unavoidable due to network asynchrony and Byzantine behavior. To mitigate this cost, Pufferfish employs a commit-aware snapshot mechanism to preserve as many valid pre-commit execution results as possible by taking snapshots without introducing heavy memory or storage overhead. Briefly, instead of eagerly snapshotting every speculative order, replicas take lightweight snapshots only when mis-speculation is likely. Once the speculative order is indeed invalid, replicas roll back to the most recent valid snapshot and re-executes only the affected transactions.

Figure 3 illustrates the overall workflow of Pufferfish. Transaction processing proceeds as follows:

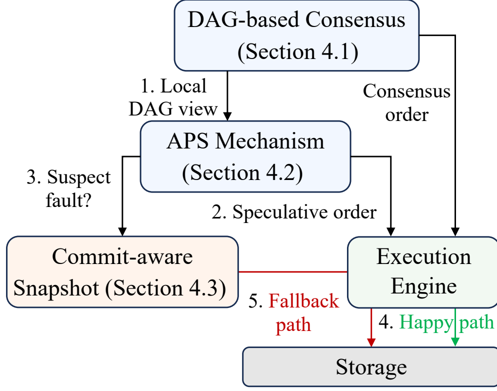


Figure 3. Overview of Pufferfish: replicas run an underlying DAG-based consensus for efficient data dissemination, predict the transaction order with the APS mechanism, and speculatively execute transactions while taking commit-aware snapshots, by which transactions are efficiently committed with speculative execution results and snapshots even during intermittent ordering failures.

1. Replicas employ Mysticeti to efficiently disseminate blocks of transactions and forward them to the APS module.
2. The APS module derives a speculative order from the local DAG view and passes it to the execution engine.
3. If the speculative order involves potentially faulty leaders, the replica takes a snapshot before execution. It then speculatively executes transactions in the predicted order.
4. (Happy path) Once consensus finalizes the order, the replica verifies consistency. If the speculative order matches, the execution results are directly committed.
5. (Fallback path) If the speculative order is inconsistent, the replica rolls back to the latest valid snapshot and re-executes only the affected transactions before committing.

4 Pufferfish Protocol

4.1 Mysticeti Consensus

Pufferfish is built on Mysticeti consensus protocol. Mysticeti disseminates transactions in blocks that form a DAG, where references between blocks encode voting information used for ordering.

DAG construction. Mysticeti proceeds in logical rounds. In each round r, every replica R_i creates a block B_i^r containing transactions and at least $2f+1$ references to round $r-1$ blocks, and broadcasts it to other replicas. Upon receiving $2f+1$ round r blocks (including its own), R_i advances to round $r+1$. Through this process, replicas collectively construct a DAG in which vertices are blocks and the edges are references.

DAG patterns and decision rules. Ordering is performed by locally interpreting the DAG. For each round, a designated leader proposes a leader block. Mysticeti defines two DAG patterns for each leader block B_L: (i) *skip pattern*, if B_L is *not* referenced by at least $2f+1$ blocks; (ii) *certificate pattern*, if

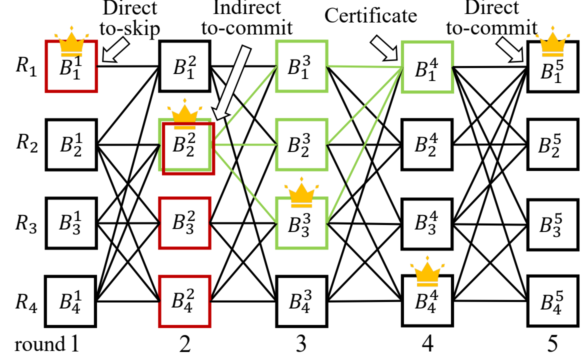


Figure 4. DAG patterns and decision rules in Mysticeti, where 👑 are leader blocks: (1) The red pattern (□) represents a *skip pattern* on B_1^1. The green pattern (□ and —) in round 3 represents a *certificate pattern* on B_2^2, and block B_1^4 is a certificate for B_2^2. (2) B_1^1 is directly decided as to-skip. Assume B_1^5 is directly decided as to-commit. Then B_2^2 is indirectly decided as to-commit from B_1^5.

B_L is referenced by at least $2f+1$ blocks. Any subsequent block that includes a certificate pattern on B_L in its causal history is called a *certificate* for B_L.

Replicas invoke an ordering instance for each leader block to commit or skip leader blocks using two decision rules:

- *Direct decision rule:* A round r leader block B_L^r is directly decided if it (i) has at least $2f+1$ certificates from round $r+2$ (decided as *to-commit*), or (ii) is identified as a *skip pattern* (decided as *to-skip*). Otherwise, it remains undecided.
- *Indirect decision rule:* For any undecided round r' leader block $B_L^{r'}$, a replica searches for the first subsequent leader block $B_L^{r''}$ (where $r''>r'+2$) that is decided as to-commit. If $B_L^{r''}$ causally references a certificate for $B_L^{r'}$, then $B_L^{r'}$ is decided as to-commit; otherwise, it is decided as to-skip.

Figure 4 illustrates an example of how leader blocks are decided in Mysticeti. The round 1 leader block B_1^1 is identified as a *skip pattern* and is directly decided as to-skip. The round 3 leader block B_3^3 is decided as to-commit by the direct decision rule, as all round 5 blocks are its certificates. However, the round 2 leader block B_2^2 is neither directly decided as to-commit (as it has only two certificates B_1^4 and B_2^4) nor directly decided as to-skip. Assume that B_1^5 is directly decided as to-commit later. Since B_1^5 causally references a certificate pattern on B_2^2 (highlighted in green), B_2^2 is decided as to-commit by the indirect decision rule.

From DAG to total order. Block ordering in Mysticeti is driven by the decision statuses of leader blocks. Once all leader blocks up to round r are decided, replicas order all *to-commit leader blocks* and their causal history. The order of causal history blocks is specified by its corresponding to-commit leader block, which can use any deterministic linearization algorithm [77]. For instance, in Figure 4, B_1^1 is decided as to-skip, while B_2^2 and B_3^3 are decided as to-commit.

Therefore, replicas will order B_2^2 and its causal history (i.e., B_2^1, B_3^1, and B_4^1) first, and then order B_3^3 and its causal history (i.e., B_3^2 and B_4^2). This results in a total order of blocks: B_2^1, B_3^1, B_4^1, B_2^2, B_3^2, B_4^2, B_3^3.

Ordering continuity. The DAG structure and indirect decision rule provide an ordering continuity property for to-commit leader blocks: subsequent to-commit leader blocks can extend the order specified by the previous to-commit leader block from its causal history [79]. For instance, in Figure 4, although B_2^2 is not directly decided (i.e., its ordering instance fails), the subsequent directly committed B_1^5 extends the order specified by B_2^2. That is, the successful ordering of B_1^5 does not invalidate the order specified by B_2^2. This feature makes the DAG-based consensus inherently suitable for our pre-commit execution scheme, since it stabilizes speculative execution by ensuring that later ordering decisions rarely invalidate earlier implied orders, thereby increasing the fraction of speculative execution results that can be directly committed (we extend a discussion in Appendix A).

4.2 Adaptive Probabilistic Speculation

The key to achieving effective pre-commit execution is to predict the final consensus order as closely as possible. To this end, we propose an adaptive probabilistic speculation (APS) mechanism. The key insight behind it is that during the data dissemination, the DAG structure encodes implicit voting signals, which can be used to estimate the likelihood of different ordering outcomes. APS leverages this information to predict a speculative order that is likely to match the final consensus order. Specifically, the APS mechanism is a two-fold design: (i) a separated prediction model speculating on the status of each leader block, and (ii) a cooperative APS tree structure enabling replicas to adaptively select the most likely speculative order.

Predict statuses of leader blocks. Recall that in Mysticeti, the order of transactions is determined by the order of leader blocks that are decided as to-commit status. Thus, to speculate on an pre-commit execution order, replicas first predict the status of each leader block.

A replica can predict the status of a leader block as soon as it receives the leader block. However, the prediction accuracy might be low at this point, since the replica has limited information about how other replicas view this leader block. On the other hand, if a replica delays the prediction until it receives enough blocks in subsequent rounds, then the prediction accuracy could be improved. However, this also delays the pre-commit execution, thereby increasing the transaction commitment latency. To balance the prediction accuracy and latency, Pufferfish employs a *one-round deferred* prediction strategy. To elaborate, a replica starts predicting the status of a round r leader block when it receives a quorum of $2f+1$ blocks from the next round $r+1$.

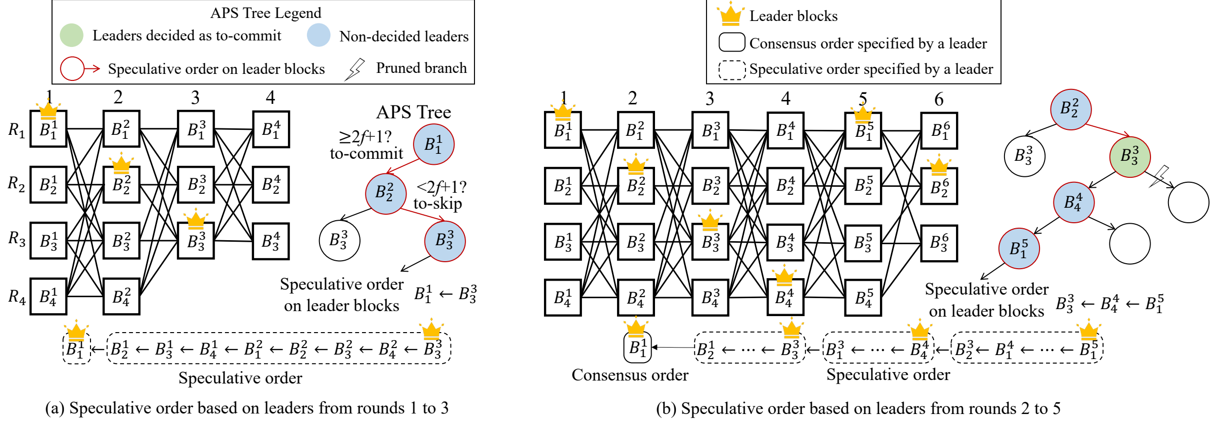
```
Variables:
  DAG_i[] - An array of sets of blocks (indexed by rounds)
  r_spl - The round of the last predicted leader block
  St_spl - The latest speculative state
  struct Leader Prediction LP:
    LP.block - the leader block
    LP.round - the round number of a pending leader block
    LP.status - the predicted status (to-commit/to-skip)
  L_spl - The APS tree, consisting of a list of LP
 1: upon receiving a quorum of 2f+1 blocks in round r+1 do
 2:     sequence ← []                        ▹ newly predicted leader blocks
 3:     LP ← ()
 4:     for r' ∈ [r down to r_spl+1] do
 5:         LP.block ← get_leader_block(r')
 6:         LP.round ← r'
 7:         LP.status ← predict_status(DAG_i[r'+1], LP.block)
 8:         sequence ← (LP)||sequence
 9:     L_spl.append(sequence)
10:     r_spl ← r
11:     St_spl ← speculative_execution(St_spl, sequence)
12: procedure predict_status(blocks, B_L)
13:     if B_L == ⊥ then
14:         return to-skip
15:     voters ← {B ∈ blocks : B_L ∈ B.parents}
16:     return |voters| ≥ 2f+1 ? to-commit : to-skip
```

Figure 5. Leader prediction with APS for replica R_i.

Figure 5 shows the leader block prediction process. When receiving a quorum of blocks in round $r+1$, a replica R_i iterates to predict all undecided leader blocks with a round $r' \leq r$ until its last predicted round r_{spl} (lines 4-8). Note that R_i performs an iteration of predictions here, as it might receive blocks from different rounds simultaneously due to the network delays, and has not predicted the status of all undecided leader blocks.

Align with Mysticeti's decision rules, Pufferfish applies a specific rule to the predictions of leader block status (lines 12-16). For round r' leader block $B_L^{r'}$, a replica R_i updates its prediction based on the number of round $r+1$ referencing it. If $B_L^{r'}$ is directly referenced by at least $2f+1$ blocks in round $r+1$ in R_i's local DAG view $DAG_i[r+1]$, $B_L^{r'}$ is predicted as to-commit; otherwise, it is predicted as to-skip. The rationale behind this prediction rule is that a leader block is decided as to-commit (via both direct and indirect decision rules) only if it is referenced by at least $2f+1$ blocks in the subsequent round. According to internal measurements from Sui mainnet, about 99% of leader blocks are committed with such signals. As we also show in Section 6, this prediction rule is effective in practice, since the referencing relationship is a strong indicator of the final decision status of a leader block.

APS tree. Once predicting the statuses of leader blocks, the replica establishes a speculative order with an *APS tree* maintained locally. The APS tree is a binary tree consisting of a list of pending leader blocks—that is, the leader blocks that have not yet been added to the consensus order. The root node of the APS tree represents the earliest undecided leader block, and each intermediate node represents a leader



(a) Speculative order based on leaders from rounds 1 to 3

(b) Speculative order based on leaders from rounds 2 to 5

Figure 6. The workflow of the APS mechanism: replica builds an APS tree to predict the speculative order, where a leader block is predicted as to-commit if it is referenced by $2f+1$ blocks. (a) The replica predicts the order of blocks from round 1 to 3. (b) The replica updates its APS tree and predicts the order of blocks from round 2 to 5.

block whose round number is proportional to the tree level. Each node has two branches: the left branch indicates that the corresponding leader block will be identified to-commit, while the right branch represents a to-skip result. Each path from the root to a leaf node represents a possible speculative execution order of leader blocks.

Workflow of the APS mechanism. Figure 6 illustrates the workflow of the APS mechanism. The left part of Figure 6(a)(b) shows a replica's local DAG view as the round progresses, while the right part shows the update of APS tree during the process. In particular, when proceeding in round 4, the replica R_1 cannot decide leader blocks B_1^1, B_2^2, and B_3^3, as neither the direct nor the indirect decision rules can yet be applied. Thus, it builds an APS tree with three levels, where the root node is B_1^1, the intermediate nodes are B_2^2 and B_3^3. With our prediction rules, R_1 predicts that B_1^1 and B_3^3 will be to-commit, while B_2^2 will be to-skip, leading to a speculative order on leader blocks $B_1^1 \leftarrow B_3^3$. As a result, R_1 derives a speculative execution order on all blocks (including non-leader blocks) according to the causal orders specified by B_1^1 and B_3^3, as shown at the bottom of Figure 6(a). Replicas continuously update their APS trees as they receive new blocks and confirm a new consensus order. In Figure 6(b), when R_1 is proceeding in round 6, B_1^1 has been directly decided as to-commit and has been removed from the APS tree. R_1 rebuilds the APS tree and predicts a new order on the pending leader blocks from rounds 2 to 5. Note that although being decided as to-commit directly in round 5, B_3^3 is still in the tree as a pending leader block, since its previous leader block B_2^2 is still undecided, making B_3^3 unable to be added to the commit sequence. Nevertheless, replicas prune the opposite branches for such decided but pending leader blocks in the APS tree, to optimize the order prediction space.

After deriving a speculative execution order, the replica forwards the order to the execution engine to perform pre-commit execution via the function speculative_execution (line 11). We defer the pseudocode of speculative_execution (Figure 7, lines 38-48) to Section 4.3 after introducing Pufferfish's snapshot mechanism.

4.3 Commit-aware Snapshots

Replicas might predict invalid speculative orders due to network asynchrony or malicious behaviors. For instance, Byzantine replicas selectively share their leader blocks to some correct replicas, making these blocks are predicted as to-commit but actually to-skip. In cases where replicas mispredict speculative orders, the pre-commit execution results will be invalid, and replicas need to re-execute all transactions. This makes the system revert to the unoptimized design and fails to mask ordering failures.

A straightforward solution to mitigate the overhead caused by mispredictions is to take snapshots of pre-commit execution states after executing on *every* leader block. This allows replicas to roll back to a recent valid state if the pre-commit execution fails. However, frequently snapshotting might introduce significant memory or storage overhead, especially during prolonged ordering stalls.

Pufferfish introduces a *commit-aware snapshot* mechanism to reduce the snapshot overhead. Specifically, instead of taking snapshots for every leader block, replicas in Pufferfish take snapshots only if they suspect an incoming speculative order is specified by a malicious replica. Our key observation is that the adversary can only mislead the prediction on its own leader blocks, and if two consecutive leader blocks are both created by correct replicas, then taking a snapshot for the later correct leader block is enough to cover the state changes occurred by the pre-commit execution on the former correct leader block. In Pufferfish, correct replicas employ a speculation-aligned reputation mechanism to identify suspect malicious replicas.

Speculation-aligned reputation. Each replica R_i in Pufferfish maintains a reputation table TR_i, where the reputation

score $TR_i[j]$ on a replica R_j reflects whether its historical leader blocks can help R_i to derive valid speculative execution orders. Specifically, R_i updates its reputation table TR_i based on the following rules (Figure 7, lines 18-23):

- *Reputation Increase:* R_i increases R_j's reputation by 1 if its prediction on the status of a leader block B_L^r created by R_j (i.e., $R_j = B_L^r.creator$) is correct.
- *Reputation Decrease:* R_i decreases R_j's reputation by 1 if its prediction on the status of a leader block created by R_j is incorrect.

Intuition behind the reputation mechanism. The reputation mechanism is designed to capture whether the replicas' behaviors are aligned with the speculation goals. A replica whose historical leader blocks often leads to correct speculative orders is likely honest, while a replica whose historical leader blocks often misleads correct replicas is likely Byzantine. Thus, correct replicas can leverage the reputation mechanism to identify potentially malicious replicas.

Snapshot policy. With the reputation mechanism, a replica R_i takes snapshots before performing pre-commit execution on leader blocks that are created by low-reputation replicas. A snapshot is a key-value pair where the key is a sequence of leader blocks that are predicted as to-commit (i.e., the speculative order), and the value is pre-commit execution states based on the speculative order (i.e., state transitions).

Replicas take snapshots when performing speculative execution (Figure 7, lines 38-48), and the decision of taking snapshots is parametrized by a threshold τ_{rep}. Specifically, upon receiving a new list of predicted leader blocks LPs, R_i extracts all predicted to-commit leader blocks in LPs to decide a speculative order (lines 40-47). R_i takes snapshots before performing pre-commit execution on leader blocks created by replicas with reputation scores at most the τ_{rep}-th lowest value in TR_i (lines 43-45). In Pufferfish, we set $\tau_{rep}=f$ since there are at most f Byzantine replicas. After that, R_i linearizes transactions in each predicted to-commit leader block and its causal historical blocks (line 46), and executes them with given states (line 47) via the execute function.

```
Variables:
  St_glb - The global key-value store state
  St_spl - The latest speculative state
  S_spl - A map from speculative leaders' statuses to snapshots
  r_lc - The last round where the leader block is committed via consensus
  TR_i[] - An array of reputations (indexed by replicas)
  L_spl - The APS tree, consisting of a list of LP
17: upon receiving an ordered leader block list L_con from consensus do
18:   for B_L ∈ L_con do                                  ▷ update reputation
19:     LP ← the entry in L_spl with LP.round = B_L.round
20:     if LP.status == to-commit then
21:       TR_i[B_L.creator] ← TR_i[B_L.creator] + 1
22:     else
23:       TR_i[B_L.creator] ← TR_i[B_L.creator] − 1
24:   if L_spl is consistent with L_con then              ▷ Happy path
25:     commit_state(St_glb, St_spl)
26:   else                                                ▷ Fallback path
27:     (St_Δ, len) ← find_bset_snapshot(L_con, S_spl)
28:     for idx = len to |L_con| − 1 do          ▷ execute inconsistent txs
29:       B_L ← L_con[idx]
30:       txs ← serialize_transactions(B_L)
31:       St_Δ ←execute(St_Δ, txs, St_glb)
32:     commit_state(St_glb, St_Δ)
33:     sequence ← the leader predictions in L_spl after L_con[−1]
34:     St_spl ← speculative_execution(St_Δ, sequence)
35:   r_lc ← L_con[−1].round                    ▷ update last consensus round
36:   update_aps_tree(L_con)                        ▷ remove decided leaders
37:   update_snapshots(S_spl, r_lc)
38: procedure speculative_execution(St_base, LPs)
39:   St_Δ ← St_base              ▷ State transition after executing on LPs
40:   for LP in LPs do
41:     if LP.status == to-commit then
42:       B_L ← LP.block
43:       if TR_i[B_L.creator] ≤ the f-th lowest value in TR_i then
44:         key ← [LPs[0], LPs[1], ..., LP]
45:         S_spl[key] ← St_Δ                       ▷ commit-aware snapshot
46:       txs ← serialize_transactions(LP)
47:       St_Δ ←execute(St_Δ, txs, St_glb)
48:   return St_Δ
```

Figure 7. Transaction commit for replica R_i in Pufferfish.

4.4 Transaction Commitment

Replicas leverage pre-commit execution results to commit transactions if the speculative order is consistent with the final consensus order. In cases where the speculative order is invalid, Pufferfish's snapshot mechanism allows replicas to quickly recover. Figure 7 shows the transaction commitment process for a replica R_i in Pufferfish. When receiving an ordered list of leader block L_{con} from the consensus layer, R_i first updates the reputation table TR_i based on the correctness of its predictions on the leader blocks in L_{con} (lines 18-23). Then, if the predicted speculative order in $\mathcal{L}_{spl}$ is consistent with the consensus order (i.e., happy path), R_i directly commits the speculative state St_{spl} to the global state St_{glb} using the function commit_state (lines 24-25). Otherwise, R_i commits transactions in a fallback path (lines 26-34). In this case, R_i finds the most recent valid snapshot (i.e., snapshot with the longest key consistent with L_{con}) in $\mathcal{S}_{spl}$, and executes all transactions in the inconsistent suffix of L_{con} to update the global state (lines 28-31). The fallback execution results are then used to update the global state (line 32). In addition, R_i will update the latest speculative state St_{spl} based on the fallback execution results (line 34), which ensures St_{spl} is always consistent with a valid order and can be used for future predictions. Finally, R_i updates its APS tree by removing the decided leader blocks (line 36), and updates snapshots by cleaning invalid ones and removing L_{con} from the prefix of the valid keys (line 37).

5 Analysis

5.1 Performance Analysis

We focus on the speedup of transaction confirmation latency achieved by Pufferfish compared to a baseline SMR system that employs Mysticeti consensus protocol but adopts an execution-after-consensus (EAC) execution scheme. The transaction confirmation latency in SMR can be measured as the time taken from when a transaction is submitted by a client to when it is committed by replicas. It consists of three primary stage latencies: (i) queuing latency t_q, indicating the time required for a transaction to be packaged into a block; (ii) consensus latency t_c, representing the time required for replicas to reach agreement on a block of transactions; and (iii) execution latency t_e, indicating the time required for replicas to execute a block of transactions in the execution engine. Note that the DAG-based consensus protocol involves two kinds of consensus latency [57], one for leader block t_{c1} and another for non-leader block t_{c2}. In the following analysis, we assume the system is not saturated in handling transactions, i.e., the number of transactions the system can handle per second is no fewer than the transaction workload. Otherwise, it will introduce additional waiting latency (e.g., a transaction is received by replicas but cannot be packaged into a block due to the exhaustion of bandwidth resources) and make the transaction confirmation latency hard to quantify.

Since the baseline adopts the EAC execution scheme, the transaction confirmation latency of the baseline system is directly derived as the summation of the three stage latencies, i.e. $t_q + t_c + t_e$, where the average consensus latency is $t_c = t_{c1}/n + (n-1)t_{c2}/n$ with n replicas and one leader node for each round in the system. In contrast, transactions in Pufferfish can be speculatively executed during the consensus, in which the execution latency t_e is overlapped by the consensus latency. The transaction confirmation latency of Pufferfish is $t_q + t_c$. As a result, the speedup of Pufferfish compared to the baseline system is $\frac{t_q+t_c+t_e}{t_q+t_c} = 1 + \frac{t_e}{t_q+t_c}$.

The latency speedup is affected by many factors. For instance, the speedup increases with the transaction workload (i.e., the number of transactions submitted by clients per second), since the transaction execution latency per block t_e increases with the transaction workload while the queuing latency t_q and consensus latency t_c are relatively stable. This has been proved in our evaluation Section 6.1. Moreover, the speedup can be higher when ordering failures happen, since the transaction execution latency t_e will increase in the baseline system due to the accumulated backlog transactions (which introduces extra waiting latency to transaction execution) during the ordering failure. This is also proved in our evaluation Section 6.2, where we observe that with the same (relatively low) transaction workload, Pufferfish achieves the confirmation latency closed to (i.e., the speedup is small) the baseline when no ordering failures occur; however, the confirmation latency of Pufferfish is much lower than that of the baseline system when ordering failures happen. This indicates that the latency speedup increases with the ordering failure rate, which is consistent with our analysis.

5.2 Security Analysis

Since Pufferfish is built on top of Mysticeti, it inherits the liveness property of Mysticeti. For safety, we consider two aspects Section 2.1: transaction order consistency and state consistency. The transaction order consistency is guaranteed by the underlying Mysticeti consensus protocol. For state consistency, note that transactions are always executed in the same order as the consensus order (Figure 7). With a deterministic execution engine (e.g., EVM [24]), this guarantees that the state of all correct replicas is consistent after executing the same sequence of transactions.

6 Evaluation

Our evaluation aims to answer the following questions:

- **Performance** How does Pufferfish perform under happy path (i.e., no faults nor network jitter) with different replicas n and under crash fault scenarios (Section 6.1)?
- **Ordering Failure Resistance** Can Pufferfish effectively resist intermittent ordering failures (Section 6.2)?

Implementation. We implement Pufferfish in Rust based on the Mysticeti codebase [1]. As Mysticeti only implements the consensus layer, we integrate an execution engine to support transaction execution. In our implementation, we choose EVM (implemented by RiseLabs [51]) as our execution engine due to its well-defined interfaces and the popularity of EVM-compatible transactions. Nevertheless, Pufferfish is agnostic to the choice of execution engine, since it optimizes transaction execution in a system-level manner rather than in an instruction-level manner. To minimize the execution benefits offered by the execution engine, we use the basic sequential execution mode of the Rise EVM in our evaluation.

One of our key aspects is to demonstrate the effectiveness of Pufferfish in resisting intermittent ordering failures caused by network jitter in practice. Although we can already observe network jitter and ordering failures in our realistic evaluation environment, they are mainly dependent on the real-time Internet conditions and are uncontrollable. This uncontrollability (i) makes it hard to conduct the same settings for fairly comparing Pufferfish with others; (ii) makes it impossible to reproduce the results; (iii) leads to unnecessary monetary cost by repeating the experiments until we observe the intermittent ordering failures. To address this issue, we add the minimum manual effort to simulate the intermittent ordering failures. Specifically, we simulate the intermittent ordering failures by randomly injecting message delay (0 to 2 seconds) to connections among replicas (we used a pseudorandom number generator with a fixed seed to ensure

the reproducibility and fair comparison) when conducting evaluations for ordering failure resistance (Section 6.2).

Baseline. We compare Pufferfish with a representative SMR protocol that employs Mysticeti as the consensus layer, since Mysticeti is a state-of-the-art protocol decoupling transaction dissemination and ordering that is deployed in production [64]. For a fair comparison, we employ the same EVM execution engine on top of the Mysticeti consensus layer and use the same transaction scheduler for both systems. Different from Pufferfish, the baseline adopts a classic execution-after-consensus (EAC) design, where transactions are scheduled to be executed only after the success of ordering. We therefore name this baseline as *Mysticeti-EAC* in the rest of this section.

Experiment setup. We evaluate all systems on AWS, using c5a.4xlarge EC2 instances spread across 5 regions (useast-1, us-east-2, us-west-1, eu-west-1, and eu-west-2). The ping latencies among regions range from 50 to 80 ms. Each instance provides 16 vCPU, 32GB RAM, and up to 10 Gbps of bandwidth and runs Linux Ubuntu server 20.04.

We deploy one client per replica to continuously send transactions in a steady way. We use ERC20 token transfer transactions as the workload, which is one of the most common transactions in most EVM-based blockchains. We focus on several performance metrics, including *throughput*, which represents the number of transactions committed per second (TPS), and *commitment latency*, which is measured by the time from when clients send transactions to when the transactions are committed to the storage. We run each experiment for at least 100 seconds until we observe the steady performance.

6.1 Performance

We first evaluate the performance of Pufferfish by running varying numbers of replicas n=10 and 49, and considering crash-free and crash fault scenarios. In the performance evaluation, we set different transaction input rates/workloads and monitor the throughput and commitment latency. We stop increasing the transaction workload when we observe a shaped latency increase, which indicates the system is saturated in handling transactions (i.e., the number of transactions the system can handle per second is lower than the transaction workload).

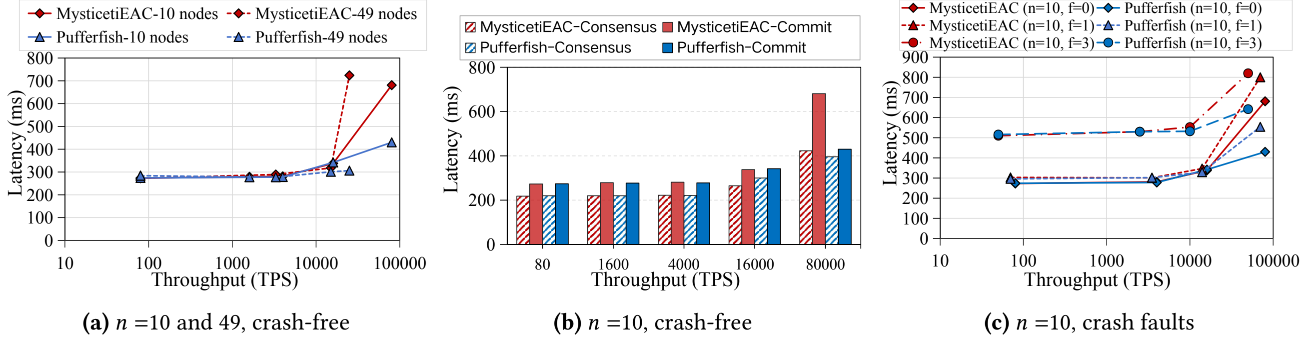
Crash-free performance. Figure 8a shows the throughput-latency curves of Pufferfish and Mysticeti-EAC with different numbers of replicas n. For both n=10 and n=49, Pufferfish achieves the consistent performance as Mysticeti-EAC before the system becomes saturated. In particular, both Pufferfish and Mysticeti-EAC can achieve a steady 16,000 TPS with a latency of around 300 ms. With n=10, we observe that Mysticeti-EAC becomes saturated when the transaction workload reaches 80,000 transactions per second, where the commitment latency increases sharply to 680 ms. In contrast, Pufferfish achieves 80,000 TPS with a latency of around 430 ms, 37% lower than Mysticeti-EAC. With n=49, we observe that Mysticeti-EAC becomes saturated when the transaction workload reaches 25,000 transactions per second, where the commitment latency increases sharply to 724 ms. In contrast, Pufferfish achieves 25,000 TPS with a latency of around 300 ms, 59% lower than Mysticeti-EAC.

Figure 8b further shows the breakdown of the commitment latency into the consensus latency (i.e., the time from when a block is created to when the block is ordered) and non-consensus latency, which includes the transaction execution latency and the transaction queuing latency (i.e., the wait time of transactions being packaged into blocks). We can see that when transaction workload is high (e.g., 80,000 TPS), the non-consensus latency of Mysticeti-EAC accounts for a significant portion of its commitment latency, while Pufferfish can still maintain a relatively low non-consensus latency. This is because Pufferfish can better utilize the execution resources by allowing pre-commit execution, leading to a lower transaction queuing latency and a lower post-consensus execution latency.

Performance under crash faults. We further evaluate the performance of Pufferfish and Mysticeti-EAC under crash faults. In particular, we set f=1 and 3 crash faults out of n replicas in total, where the crash faults are completely unresponsive while the system is running. Figure 8c shows the throughput-latency curves of Pufferfish and Mysticeti-EAC with different numbers of crashed replicas f. We can see that both Pufferfish and Mysticeti-EAC can maintain a similar throughput-latency curve as the crash-free scenario when f is small (e.g., f=1). However, when f becomes larger (e.g., f=3), we observe a significant latency increase. This is because the existence of crash faults will lead to timeouts when they become leaders, in which other replicas have to wait for the timeout before moving to the next round. Nevertheless, Pufferfish can achieve a better performance than Mysticeti-EAC under crash faults.

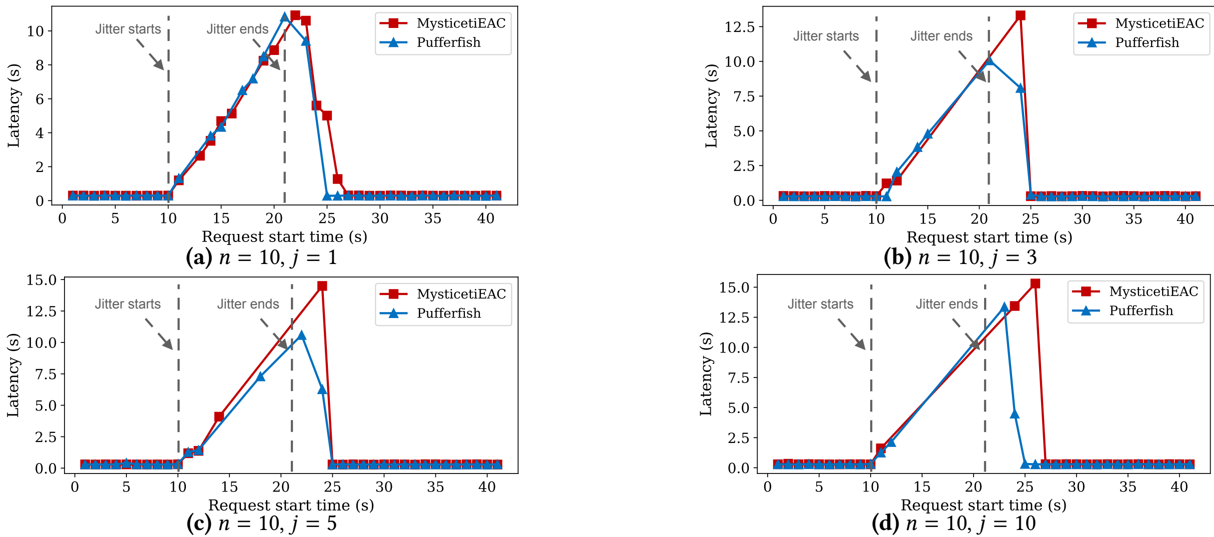
6.2 Ordering Failure Resistance

We then evaluate whether Pufferfish can effectively resist intermittent ordering failures caused by network fluctuation/jitter by running n=10 replicas with a transaction workload of 16,000 transactions per second. To simulate network jitter, we randomly inject message delays (0 to 2 seconds) to connections among j replicas. We set j=1, 3, 5, and 10 to comprehensively reflect different levels of network fluctuation in practice. In the evaluation, these j replicas experience the network jitter starting at roughly 10 seconds and ending at roughly 21 seconds. We record the transaction latency every second, each data point indicates that the system can commit and execute transactions within that second.



(a) n =10 and 49, crash-free **(b)** n =10, crash-free **(c)** n =10, crash faults

Figure 8. Performance evaluation of Mysticeti-EAC and Pufferfish: (a) crash-free performance with different numbers of replicas n; (b) crash-free consensus latency and non-consensus latency with n =10; (c) performance under crash faults.



(a) n = 10, j = 1 **(b)** n = 10, j = 3 **(c)** n = 10, j = 5 **(d)** n = 10, j = 10

Figure 9. Transaction commitment latency when different numbers of replicas experience network fluctuation.

Figure 9 shows the transaction latency of Pufferfish and Mysticeti-EAC when different numbers of replicas experience network jitter. Intuitively, both Pufferfish and Mysticeti-EAC will experience a latency increase during the network jitter period, as the network jitter can lead to intermittent ordering failures, which introduce higher consensus latency. In addition, both systems take time to recover after the network jitter period (i.e., they can commit transactions right after the jitter ends), and with more replicas experiencing the network jitter, the recovery time increases. However, Pufferfish can recover faster than Mysticeti-EAC after the network jitter period. For instance, when j=3, baseline recovers and resume committing transactions at the 24 seconds mark, while Pufferfish can recover at 21 seconds.

This is because Pufferfish better utilizes its execution resources to pre-execute transactions even during the intermittent ordering failures and therefore takes less time to handle the accumulated backlogs of transactions (where there pre-execution results can be directly applied). In contrast, Mysticeti-EAC waits for the success of ordering before executing transactions: its execution-after-consensus design means replicas execute all backlog of transactions while new transactions wait. Moreover, thanks to the fast recovery from intermittent ordering failures, Pufferfish can maintain a relatively low transaction commitment latency on average and achieve a lower tail (i.e., p99) transaction confirmation latency compared to Mysticeti-EAC. For example, when j=5, the tail commitment latency of Mysticeti-EAC is around 15s (at 24 seconds), while the tail commitment latency of Pufferfish is around 11s (at 22 seconds), a 1.36x improvement.

7 Related Work

Speculative execution. The key technique employed by Pufferfish is pre-commit execution, a closed concept to speculative execution that has been widely adopted to optimize transaction confirmation latency across all layers of computer systems [9, 15, 34, 38, 41, 43, 45–47, 50, 65, 68, 69, 72, 76, 80]. Among them, Forerunner [15] and its successors [47, 80] apply speculative execution to BFT SMR by speculatively executing *local* transactions before they are disseminated. These

designs primarily target Ethereum's coupled BFT consensus [11], and thus do not address the limitations of data dissemination. Moreover, speculative execution on local transactions lacks global ordering signals, which can lead to a high rate of mis-speculation due to inaccurate execution orders. More recent works, including Hotstuff-1 [35] and Zaptos [69], adopt speculative execution in decoupled BFT designs. They pre-execute transactions using heuristics derived from block proposals, improving speculation accuracy under common cases. However, these works do not provide order continuity: when ordering failures occur, replicas perform view changes that discard previously failed ordering proposals and replace them with new proposals. As a result, speculative execution results from failed ordering proposals are invalidated, and execution must restart. Consequently, these approaches improve latency only in common cases and cannot mask intermittent ordering failures. The concurrent work MonadBFT [33] introduces a tail-forking resistance property to prevent transaction reorder issues such as MEV [19]. Although it is not designed for masking ordering failures, we observe that it can incidentally preserve speculative executions in some cases, where leaders are timely responsive or all correct replicas are not affected by network fluctuations, making it more sensitive to network instability and leader availability. In contrast, Pufferfish directly targets intermittent ordering failures by leveraging DAG-based consensus to achieve ordering continuity without relying on leader availability or additional coordination mechanisms. Furthermore, all these speculative execution [33, 35, 69] are designed for Hotstuff-style consensus, which typically achieves 5Δ transaction confirmation latency in common cases, while Pufferfish, built on Mysticeti, can achieve 3Δ latency, which makes it the best candidate for future blockchains.

Deconstructed SMR schemes. A line of work [3, 30, 44, 48, 49, 52, 55, 56, 73, 78] attempts to reduce the overhead introduced by ordering failures by deconstructing the SMR scheme. HyperLedger [3] and its successor [52, 55] adopt an execution-order-validation scheme, which allows replicas to execute transactions asynchronously even when ordering fails. Fabric SSI [44] and Bidl [49] employ an execute-parallel-order-validate scheme that hides the ordering cost by parallelizing the execution and ordering tasks. However, all these schemes (partially) deconstructing schemes still couple the data dissemination with ordering/execution, which is instead observed as a main bottleneck for achieving high performance [7, 20, 28, 75, 78]. Recent works [17, 56, 78] use a sharded-disperse-order-execution scheme to achieve efficient transaction dissemination regardless of ordering failures. However, these works use a latent execution-after-consensus scheme, which still suffers from a significant execution overhead due to the backlog of transactions when intermittent ordering failures occur.

Pipelined SMR architectures. Many systems [5, 12, 14, 21, 32, 35, 42, 54, 69, 70, 74] have adopted pipelined architectures to optimize the transaction confirmation latency for BFT SMR, which divide the handling of transactions into multiple stages and pipeline them to reduce the overall latency. However, these pipelined architectures only work as expected under common cases (i.e., no ordering failures), since an ordering failure can make all the pipelined stages invalid. In contrast, Pufferfish is designed to optimize the transaction confirmation latency under both common cases and ordering failures.

Parallel transaction execution. An orthogonal line of work optimizes transaction execution by concurrent/parallel transaction execution [4, 13, 16, 26, 31, 37, 40, 53, 67, 71]. These works study concurrent strategies, transaction scheduling algorithms, and parallel execution engines to allow a single replica to execute transactions in parallel, thereby reducing the transaction execution latency. Since Pufferfish is a system-level optimization scheme, these parallel execution techniques can be integrated into Pufferfish to further improve the transaction throughput and latency.

8 Conclusion and Discussion

This work presents Pufferfish that aims to optimize the performance under intermittent ordering failures for SMR via proactive pre-commit execution. Extensive evaluation results conducted on a geo-distributed environment show Pufferfish can achieve faster recovery and lower transaction confirmation latency compared to the SOTA BFT SMR system.

Powerful network adversary. Replicas in Pufferfish predict the pre-commit execution order using the heuristic based on their local views of the DAG, whose effectiveness will be affected by the messages (i.e., blocks) they receive. If the adversary has strong power to control the network and arbitrarily schedule the message delivery, then it might cause mispredictions, which can lead to a high speculation failure rate. Unfortunately, it appears impossible to completely prevent the influence of such powerful network adversaries. However, we argue that this is a strong assumption for the adversary, and in practice, most validators are well-behaved, and the network adversary usually has limited power to control the network. Therefore, we believe that Pufferfish is effective when deployed in practice.

Overwritten pre-commit execution states. Continuous pre-commit execution improves resource utilization but may cause overwritten states. For instance, if a replica has performed pre-commit execution on top of two consecutive leader blocks B_L^1 and B_L^2, and B_L^1 is decided but B_L^2 is not, then the latest speculative execution states (i.e., St_{spl} in Figure 7) cannot be committed as they might include the overwritten states by the speculative execution of B_L^2. To alleviate this issue, we currently introduce a snapshot window with a

constant size s_w that contains snapshots of pre-commit execution states for the latest s_w leader blocks. This allows replicas to commit transactions with these recent snapshots if a non-latest leader block is decided. While more sophisticated designs are possible, e.g., a fine-grained commitment mechanism that selectively commits non-conflicting transactions across consecutive leader blocks, our evaluation shows that the snapshot window mechanism already achieves strong performance in practice. Exploring these designs is orthogonal to our focus on demonstrating Pufferfish's effectiveness to circumvent performance degradation caused by ordering failures, and we leave them to future work.

References

[1] Alberto Sonnino. Mysticeti codebase. github.com/asonnino/mysticeti, 2026.

[2] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. The bedrock of byzantine fault tolerance: A unified platform for {BFT} protocols analysis, implementation, and experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 371–400, 2024.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[4] Parwat Singh Anjana, Matin Amini, Rohit Kapoor, Rahul Parmar, Raghavendra Ramesh, Srivatsan Ravi, and Joshua Tobkin. Efficient parallel execution of blockchain transactions leveraging conflict specifications. In *7th Conference on Advances in Financial Technologies (AFT 2025)*, pages 29–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

[5] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput {DAG}{BFT} can be fast and robust! In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 813–826, 2025.

[6] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the limits of latency with uncertified dags. In *Network and Distributed Systems Security Symposium (NDSS)*, 2025.

[7] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 585–602, 2019.

[8] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: Bft systems made robust. In *25th International Conference on Principles of Distributed Systems*, 2022.

[9] Jonathan Behrens, Anton Cao, Cel Skeggs, Adam Belay, M Frans Kaashoek, and Nickolai Zeldovich. Efficiently mitigating transient execution attacks using the unmapped speculation contract. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1139–1154, 2020.

[10] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 2606–2620, 2024.

[11] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper. *arXiv preprint arXiv:2003.03052*, 2020.

[12] Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[13] Junchao Chen, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. Thunderbolt: Concurrent smart contract execution with non-blocking reconfiguration for sharded dags. *arXiv preprint arXiv:2407.09409*, 2024.

[14] Wuhui Chen, Ding Xia, Zhongteng Cai, Hong-Ning Dai, Jianting Zhang, Zicong Hong, Junyuan Liang, and Zibin Zheng. Porygon: Scaling blockchain via 3d parallelism. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1944–1957. IEEE, 2024.

[15] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 570–587, 2021.

[16] Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers. *Proceedings of the VLDB Endowment*, 17(10):2541–2554, 2024.

[17] Shir Cohen, Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Proof of availability and retrieval in a modular blockchain architecture. In *International Conference on Financial Cryptography and Data Security*, pages 36–53. Springer, 2023.

[18] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483. IEEE, 2021.

[19] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *IEEE S&P*, pages 910–927, 2020.

[20] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *ACM EuroSys*, pages 34–50, 2022.

[21] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing block period and commit latency in chain-based rotating leader bft. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 470–482. IEEE, 2024.

[22] Sisi Duan, Haibin Zhang, Xiao Sui, Baohan Huang, Changchun Mu, Gang Di, and Xiaoyun Wang. Dashing and star: Byzantine fault tolerance with weak certificates. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 250–264, 2024.

[23] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

[24] Ethereum Fundation. Ethereum virtual machine (evm). https://ethereum.org/en/developers/docs/evm/. Accessed: 2026.

[25] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *FC*, pages 296–315. Springer, 2022.

[26] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Blockstm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

[27] Frank Christian Geyer, Hans-Arno Jacobsen, Ruben Mayer, and Peter Mandl. An end-to-end performance comparison of seven permissioned blockchain systems. In *Proceedings of the 24th international middleware conference*, pages 71–84, 2023.

[28] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. *ACM SOSP*, 2024.

[29] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, and Gauthier Voron. Stabl: The sensitivity of blockchains to failures. In *Proceedings of the 26th International Middleware Conference*, pages 202–214, 2025.

[30] Shengjie Guan, Rongkai Zhang, Qiuyu Ding, Mingxuan Song, Zhen Xiao, Jieyi Long, Mingchao Wan, Taifu Yuan, and Jin Dong. Nexus: A novel transaction processing framework for permissioned blockchain. *IEEE Transactions on Parallel and Distributed Systems*, 2026.

[31] Yaron Hay and Roy Friedman. Batch-schedule-execute: on optimizing concurrent deterministic scheduling for blockchains. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 163–174. IEEE, 2024.

[32] Mohammad M Jalalzai, Jianyu Niu, Chen Feng, and Fangyu Gai. Fast-hotstuff: A fast and resilient hotstuff protocol. *arXiv preprint arXiv:2010.11454*, 2020.

[33] Mohammad Mussadiq Jalalzai, Kushal Babel, Jovan Komatovic, Tobias Klenze, Sourav Das, Fatima Elsheimy, Mike Setrin, John Bergschneider, and Babak Gilkalaye. Monadbft: Fast, responsive, fork-resistant streamlined consensus. *arXiv preprint arXiv:2502.20692*, 2025.

[34] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, Taebum Kim, and Byung-Gon Chun. Speculative symbolic graph execution of imperative deep learning programs. *ACM SIGOPS Operating Systems Review*, 53(1):26–33, 2019.

[35] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. Hotstuff-1: Linear consensus with one-phase speculation. *Proceedings of the ACM on Management of Data*, 3(3):1–29, 2025.

[36] Tasos Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Ilya Sergey, Alberto Sonnino, Mingwei Tian, and Jianting Zhang. Beluga: Block synchronization for bft consensus protocols. *arXiv preprint arXiv:2511.15517*, 2025.

[37] Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. Pilotfish: Distributed execution for scalable blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 287–306. Springer, 2025.

[38] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[39] Andrei Lebedev and Vincent Gramoli. Blockchain communication vulnerabilities. *arXiv preprint arXiv:2603.02661*, 2026.

[40] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. Parallelevm: Operation-level concurrent transaction execution for evm-compatible blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 211–225, 2025.

[41] Lun Liu, Todd Millstein, and Madanlal Musuvathi. Accelerating sequential consistency for java with speculative compilation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–30, 2019.

[42] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[43] James W Mickens, Jonathan R Howell, Jacob R Lorch, Jeremy E Elson, and Edmund B Nightingale. Network application performance enhancement using speculative execution, March 20 2012. US Patent 8,140,646.

[44] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: design and implementation of a blockchain relational database. *Proceedings of the VLDB Endowment*, 12(11):1539–1552, 2019.

[45] Naveen Neelakantam, David R Ditzel, and Craig Zilles. A real system evaluation of hardware atomicity for software speculation. *ACM Sigplan Notices*, 45(3):29–38, 2010.

[46] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review*, 39(5):191–205, 2005.

[47] Rui Pan, Chubo Liu, Guoqing Xiao, Mingxing Duan, Keqin Li, and Kenli Li. An algorithm and architecture co-design for accelerating smart contracts in blockchain. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[48] Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, and Ge Yu. Neuchain: a fast permissioned blockchain system with deterministic ordering. *Proceedings of the VLDB Endowment*, 15(11):2585–2598, 2022.

[49] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, pages 18–34, 2021.

[50] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. Challenging sequential bitstream processing via principled bitwise speculation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 607–621, 2020.

[51] RiseLabs. Rise parallel evm. https://github.com/risechain/pevm, 2026.

[52] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 543–557, 2020.

[53] Donghyeon Ryu and Chanik Park. Toward high-performance blockchain system by blurring the line between ordering and execution. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2024.

[54] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A high-throughput blockchain over mobile devices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 567–582, 2020.

[55] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, pages 105–122, 2019.

[56] Nibesh Shrestha and Aniket Kate. Towards improving throughput and scalability of dag-based bft smr. In *ACM EuroSys*, 2026.

[57] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving latency of dag-based bft. In *IEEE S&P*, 2025.

[58] Nibesh Shrestha, Qianyu Yu, Aniket Kate, Giuliano Losa, Kartik Nayak, and Xuechao Wang. Optimistic, signature-free reliable broadcast and its applications. *arXiv preprint arXiv:2505.02761*, 2025.

[59] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *NSDI*, volume 8, pages 189–204, 2008.

[60] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. In *International Conference on Financial Cryptography and Data Security*, page 92–109. Springer, 2024.

[61] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *ACM CCS*, pages 2705–2718, 2022.

[62] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 17–33, 2022.

[63] Sui Engineering Team. Private communication on the June 2025 sui testnet incident, 2025. Private communication with the Sui engineering team, June 2025.

[64] The Sui Team. The sui blockchain. https://sui.io/. Accessed: 2026.

[65] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–73, 2010.
[66] Andrei Tonkikh, Balaji Arun, Zhuolun Xiang, Zekun Li, and Alexander Spiegelman. Raptr: Prefix consensus for robust high-performance bft. *arXiv preprint arXiv:2504.18649*, 2025.
[67] Hao Wang, Minghao Pan, and Jiaping Wang. Crystality: a programming model for smart contracts on parallel evms. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 412–425, 2025.
[68] Yang Xia, Peng Jiang, and Gagan Agrawal. Scaling out speculative execution of finite-state machines with parallel merge. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 160–172, 2020.
[69] Zhuolun Xiang, Zekun Li, Balaji Arun, Teng Zhang, and Alexander Spiegelman. Zaptos: Towards optimal blockchain latency. *arXiv preprint arXiv:2501.10612*, 2025.
[70] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment*, 14(11):2314–2326, 2021.
[71] Tianjing Xu, Yongqi Zhong, Yiming Zhang, Ruofan Xiong, Jingjing Zhang, Guangtao Xue, and Shengyun Liu. Vegeta: Enabling parallel smart contract execution in leaderless blockchains. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 795–811, 2025.
[72] Xinan Yan, Arturo Pie Joa, Bernard Wong, Benjamin Cassell, Tyler Szepesi, Malek Naouach, and Disney Lam. Specrpc: A general framework for performing speculative remote procedure calls. In *Proceedings of the 19th International Middleware Conference*, pages 266–278, 2018.
[73] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267, 2003.
[74] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM symposium on principles of distributed computing*, pages 347–356, 2019.
[75] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. Ohie: Blockchain scaling made simple. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 90–105. IEEE, 2020.
[76] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W Fletcher. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 707–720. IEEE, 2020.
[77] Jianting Zhang and Aniket Kate. No fish is too big for flash boys! frontrunning on dag-based blockchains. In *2025 IEEE Annual Computer Security Applications Conference (ACSAC)*, pages 1065–1080. IEEE, 2025.
[78] Jianting Zhang, Zhongtang Luo, Raghavendra Ramesh, and Aniket Kate. Optimal sharding for scalable blockchains with deconstructed smr. *Proceedings of the VLDB Endowment*, 2025.
[79] Jianting Zhang, Sen Yang, Alberto Sonnino, Sebastián Loza, and Aniket Kate. Lifefin: Escaping mempool explosions in dag-based bft. *arXiv preprint arXiv:2511.15936*, 2025.
[80] Shijie Zhang, Ru Cheng, Xinpeng Liu, Jiang Xiao, Hai Jin, and Bo Li. Seer: Accelerating blockchain transaction execution by fine-grained branch prediction. *Proceedings of the VLDB Endowment*, 18(3):822–835, 2024.

A Decoupled BFT Consensus under Ordering failures

In this section, we compare how the representative decoupled BFT consensus protocols perform the ordering task and discuss why we choose to use the DAG-based BFT consensus in Pufferfish. We focus on the partially synchronous consensus and the context where ordering failure happens, as the goal of this work is to illustrate and circumvent the performance degradation caused by intermittent ordering failures. Figure 10 illustrates three types of decoupled BFT consensus under ordering failures.

Batch-based BFT protocols (Figure 10(a)) decouple data dissemination from ordering by allowing replicas to propagate transactions in batches independently of ordering transactions. Replicas might adopt different data dissemination approaches, such as direct dissemination [25, 35, 42, 74] or certified dissemination [56, 66, 69, 78], where replicas propagate batches of full transactions to other replicas and collect a quorum of acknowledgments certifying that the corresponding full transactions have been received. During the data dissemination, replicas store the received batches of transactions $bt_1, bt_2, \cdots, bt_n$ in their mempool, as shown at the top of Figure 10(a).

Batch-based BFT consensus protocols typically employ a chain-based ordering protocol (e.g., Hotstuff-style consensus) to establish a global order for transactions with the disseminated batches, as shown in the middle of Figure 10(a). Specifically, replicas continuously invoke ordering instances to generate ordering proposals that imply orders among transactions. For each ordering instance, a designated leader replica is responsible for proposing an ordering proposal OP that contains a list of transaction batches (using their digests D_{bt}), with indices implying the order among batches. This dissemination-ordering separation (i) allows replicas to propagate transactions even during asynchrony, and (ii) enhances the communication efficiency in the ordering task since replicas only need to process lightweight metadata instead of full transactions.

The ordering instance will proceed successfully under benign conditions (including the leader replica is correct, and the network is synchronous), in which its associated ordering proposal will be committed (e.g., the green ordering proposals OP_1, OP_2, and OP_4 in Figure 10(a)). In case the ordering instance fails, replicas will trigger a *view change* to select a new leader replica and proceed with a new ordering proposal. In this case, the failed ordering proposal will be discarded. For instance, in Figure 10(a), ordering failure occurs when replicas proceed with the ordering proposal OP_3, and a new ordering proposal OP_4 is created via the view change, leading to the order specified by OP_3 (i.e., batch bt_4 is ordered after batch bt_3) being abandoned.

Since ordering failures always lead to invalid ordering proposals, these batch-based BFT consensus protocols fail to

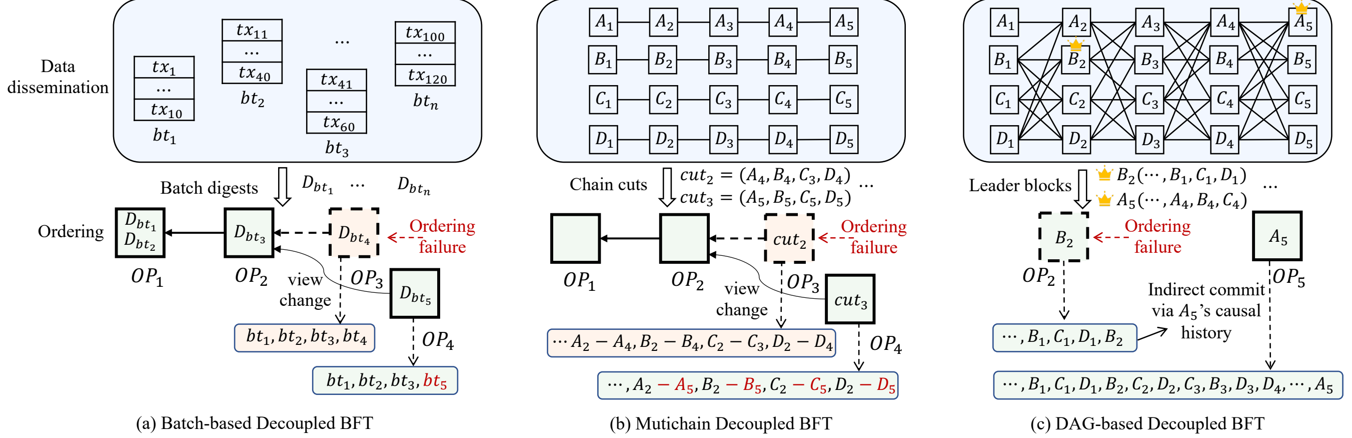


Figure 10. Comparison of different decoupled BFT consensus under ordering failures. The top demonstrates the data dissemination approach, and the button demonstrates the transaction orders specified by different ordering proposals OP.

circumvent the latency increase introduced by intermittent ordering failures with speculative execution.

Muitichain BFT protocols (Figure 10(b)) decouple data dissemination from ordering by allowing replicas to build their own chains of transaction blocks independently of ordering transactions. With the dissemination approach in multichain BFT, every replica can package transactions received from clients into blocks and append these blocks to its maintained chain in parallel. This progress typically involves a simple propose-vote scheme [28]: replicas first broadcast their blocks, and the others vote for the blocks if they have received all the history blocks of the chain extended by these new blocks. This ensures that all blocks that have been appended to chains are available.

With the available chains of blocks constructed by the data dissemination, multichain BFT consensus protocols perform the ordering task by using only the latest appended block of every chain (called *chain cuts*) as the ordering proposal. For instance, in Figure 10(b), $cut_2 = (A_4, B_4, C_3, D_4)$ is used in an ordering proposal OP_3 to specify the order among uncommitted blocks A_2, A_3, A_4, B_2, B_3, B_4, C_2, C_3, and D_2, D_3, D_4. With the chain cuts, blocks are ordered based on a deterministic rule, e.g., order based on replica identities and block heights. This enables multichain BFT consensus protocols to achieve more efficient communication in the ordering task compared to batch-based BFT, since each ordering proposal contains at most n block digests.

Unfortunately, multichain BFT consensus protocols have the same issue when ordering failures occur, since they employ a similar ordering protocol as the batch-based BFT. To elaborate, different chain cuts might imply different orders among blocks. When ordering failures occur, their inherent view change mechanism will select a new ordering proposal containing a new chain cut, which will discard the order specified by the failed ordering proposal. For instance, the ordering proposal OP_3 in Figure 10 is abandoned when it experiences an ordering failure; the new ordering proposal OP_4 containing a new chain cut will invalidate the order specified by OP_3. As a result, similar to batch-based BFT, these multichain BFT consensus protocols fail to circumvent the latency increase introduced by intermittent ordering failures with speculative execution.

DAG-based BFT protocols (Figure 10(c)) decouple data dissemination from ordering by allowing replicas to propagate blocks of transactions in logical rounds independently of ordering transactions. Specifically, each replica in DAG-based BFT propagates blocks of transactions to form a directed acyclic graph. Each disseminated block is labeled with a round number and has to reference a quorum $2f+1$ of blocks from the previous round.

Replicas in DAG-based BFT consensus protocols perform the ordering task by interpreting the DAG. To achieve that, these protocols designate leader replicas whose proposed blocks in the specific rounds are considered as leader blocks. Replicas then check whether the leader blocks receive enough votes/references from subsequent rounds (see Section 4.1 for more details). The leader blocks will specify orders among blocks in their causal histories, and will be used as ordering proposals to establish a global order.

Figure 10(c) demonstrates how blocks/transactions are ordered in Mysticeti [6]. For illustration purposes, we consider Mysticeti's single-leader, non-pipeline mode, where a leader replica is designated every three rounds, e.g., the round 2 block B_2 and round 5 block A_5 are leader blocks, and are used as ordering proposals OP_2 and OP_5, respectively. B_2 specifies an order among its causal history (i.e., B_1, C_1, and D_1), but its ordering proposal OP_2 fails to be committed directly, since there are not enough certificates in round 4 voting for it (see the direct decision rule in Section 4.1).

However, unlike the other decoupled BFT consensus protocols that abandon and invalidate the failed ordering proposal, DAG-based BFT consensus could extend the failed ordering proposal due to its order continuity property. In Figure 10(c), when the ordering proposal OP_5 is committed, its associated leader block A_5 will extend the order specified by B_2, thereby not invalidating the failed ordering proposal OP_2. This property makes DAG-based BFT consensus extremely suitable for our pre-commit execution scheme and scenarios, as it allows replicas to maintain more valid speculative execution results even when ordering failures occur.