Thunderbolt: Causal Concurrent Consensus and Execution

Junchao Chen
Exploratory Systems Lab
University of California, Davis

Alberto Sonnino
Mysten Labs
University College London (UCL)

Lefteris Kokoris-Kogias
Mysten Labs
IST Austria

Mohammad Sadoghi
Exploratory Systems Lab
University of California, Davis

Abstract

In the realm of blockchain, smart contracts have achieved widespread adoption due to their inherent programmability. However, smart contracts suffer from long execution delays, resulting from the analysis of the contract code. Consequently, the development of a system capable of facilitating high throughput and scalability holds paramount importance. Sharding represents a prevalent technique that enhances performance by horizontally scaling storage into individual shards. However, existing sharding methods rely on two-phase commit (2PC) to handle cross-shard transactions through data locking, necessitating the provision of read/write sets in advance, which poses impractical challenges for smart contracts.

This paper introduces Thunderbolt, a novel sharding architecture that integrates the Execute-Order-Validate (*EOV*) and Order-Execute (*OE*) models to manage single-shard transactions (Single-shard TX*s*) and cross-shard transactions (Cross-shard TX*s*) without leveraging 2PC to coordinate the transactions. Shards in Thunderbolt share all the replicas, and Thunderbolt assigns each replica as the shard submitter to propose the transactions. Each shard submitter employs the *EOV* model to execute Single-shard TX*s* concurrently while applying the *OE* model for executing Cross-shard TX*s*. We leverage the DAG-based protocol as the consensus protocol and modify the consensus logic to ensure correctness between Single-shard TX*s* and Cross-shard TX*s*. We implemented a concurrent executor to execute the Single-shard TX*s* locally to dynamically assign the scheduling order without any read/write set knowledge. Additionally, we introduce a novel shard reconfiguration to withstand censorship attacks by relocating the shards from the current DAG to a new DAG and rotating the shard submitters. Our comparison of the results on SmallBank with serial execution on Narwhal-Tusk revealed a remarkable 50x speedup with 64 replicas.

Keywords

concurrency control, two-phase commit, smart contract

ACM Reference Format:
Junchao Chen, Alberto Sonnino, Lefteris Kokoris-Kogias, and Mohammad Sadoghi. 2024. Thunderbolt: Causal Concurrent Consensus and Execution. In . ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The rise of blockchains has generated significant interest in the development of resilient systems capable of processing data and transactions in the presence of Byzantine behavior, such as faulty behavior originating from software, hardware, network failures, or coordinated malicious attacks [6, 8, 23, 30, 39, 56]. These systems are particularly attractive as they offer resilience across multiple independent participants [16, 26, 42, 44, 54, 55, 57, 59, 83]. Once transactions are finalized, they achieve immutability, enabling validation by any party. This aspect provides a robust security guarantee in an environment where trust cannot be ensured.

Smart contracts refer to a set of rules that function on a blockchain platform, allowing developers to create and implement solutions for various real-world challenges [15, 50]. Despite their numerous advantages, smart contract technologies encounter extended execution delays due to the analysis of the contract code utilized in running the contract programs [5]. As a result, researchers are actively exploring methods to enhance the performance of contract-based blockchain systems.

There are several approaches to improving blockchain systems. One way is to enhance transaction processing. Most blockchain systems use the Order-Execute (*OE*) model to execute transactions after reaching a consensus. These systems leverage deterministic concurrency controls by creating a dependency graph between transactions to improve parallelism [24, 60, 82, 84]. In contrast, systems like Hyperledger [8] follow the Execute-Order-Validate (*EOV*) model to increase their throughputs, which moves the execution ahead of the consensus to enhance flexibility and apply various concurrency control protocols, such as Optimistic Concurrency Control (OCC) [49],

Another direction for improvement is to enhance blockchain scalability to enable parallel execution [4, 20, 37, 41, 48, 73, 88]. These methods use sharding to reduce the cost of consensus protocols, as different shards can process transactions simultaneously. However, it is essential for sharding systems to handle cross-shard transactions (Cross-shard TX*s*) to allow users to interact with multiple shards atomically. This requires an atomic commitment protocol (ACP) to ensure consistency among global transactions [12, 13]. One widely used ACP is the two-phase commit protocol (2PC) where the coordinator asks all participants to prepare for the commitment

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and then decides whether to commit or abort the transaction based on the voting results [31].

Regrettably, most existing sharding systems that utilize 2PC to atomically execute Cross-shard TX*s* are primarily focused on UTXO-based models [4, 40]. This approach necessitates transactional locking for a duration, resulting in higher latency and reduced throughput. Furthermore, 2PC mandates that transactions provide their read/write sets in advance for the locking, rendering it unsuitable for smart contracts with Turing-complete features, where pre-analysis is impractical. As a consequence, current sharding systems are predominantly tailored for Single-shard TX*s* or rely on UTXO-based data models to address contention. This deviates from the offerings of traditional distributed databases, which furnish ACID-compliant data and transaction processing [32] adaptable to diverse application-specific requirements. This prompts the question of whether such adaptable capabilities for Cross-shard TX*s* can be extended to applications without prior knowledge of read/write sets.

In this paper, we present Thunderbolt, a novel architecture that combines the *OE* and *EOV* models to facilitate the processing of Single-shard TX*s* and Cross-shard TX*s* without the need for locking mechanisms. Similar to conventional sharding systems, Thunderbolt separates transactions into distinct shards, each of which will be proposed by a submitter, called a *shard submitter*, to prevent potential conflicts. However, unlike traditional sharding systems, where each shard uses an individual consensus protocol to finalize transactions, Thunderbolt employs a directed acyclic graph (DAG) consensus protocol [9, 10, 19, 46, 47, 52, 71, 74–76] to agree on the execution results from each shard submitter.

Thunderbolt employs the *EOV* model to disseminate the outcomes of Single-shard TX*s* to various shards while it utilizes the *OE* model to ensure the atomic commitment of Cross-shard TX*s*. Since Thunderbolt delays the execution of Cross-shard TX*s* after the consensus, Thunderbolt coordinates the execution order between Single-shard TX*s* and Cross-shard TX*s* to ensure the correctness of the execution.

Thunderbolt incorporates a concurrent executor (*CE*) to facilitate the execution of Single-shard TX*s* before reaching a consensus. Unlike traditional concurrency protocols that are based on locking [3, 25, 77] and do not allow transactions to be committed if they have not obtained all the locks; the *CE* provides a nondeterministic ordering based on transaction execution states. *CE* enables Thunderbolt to commit a transaction without requiring the acquisition of all the locks.

Additionally, Thunderbolt leverages round-robin scheduling to periodically rotate the shard submitters to enhance the system's security and liveness, or do so on demand if a shard submitter cannot propose transactions in a limited time. Thunderbolt leverages the properties of the DAG to transition the current DAG to a new one without a hard stop, facilitating the rotation of submitters for each shard.

In summary, this paper makes the following contributions.

- To our knowledge, Thunderbolt is the first sharding consensus combining *OE* and *EOV* models via leveraging a DAG-based protocol.
- A novel concurrency paradigm that introduces parallel preplay and parallel validation model for Single-shard TX*s* (concurrent consensus execution). Thunderbolt preplays Single-shard TX*s* followed by parallel verification without the need to know the read/write set in advance.
- A coordinator-free Cross-shard TX*s* agreement that Thunderbolt leverages the non-determined leader from the DAG to determine the order of the Cross-shard TX*s*.
- Live shard reconfiguration protocol that allows rotating shard assignment without the need to pause either DAG dissemination or consensus layer.
- We implemented a current executor to improve the parallelism of executing smart contracts without knowing the read/write sets on the Single-shard TX*s*. The execution engine dynamically arranges the transactions based on the current assessments to reduce the abortion rates due to the conflicts.
- Our evaluation of Thunderbolt yields a remarkable 50x speedup over Tusk [19] with a sequential execution using SmallBank workload on 64 replicas built on Apache ResilientDB (Incubating) [1, 36].

2 Preliminaries

In this section, we introduce the preliminaries on which Thunderbolt relies.

2.1 Cross-shard Transactions in Blockchain

Omniledger [48] represents an advanced blockchain system that employs sharding to facilitate Cross-shard TX*s* on the UTXO with 2PC-based protocols. The process involves initiating the transaction across all input shards to obtain individual responses and subsequently confirming the transaction in the output shards if all input shards indicate affirmation.

Regrettably, the management of smart contracts within sharding-based blockchains is currently the focus of only a limited number of systems. The Cross-Shard Function Call [63, 64] proposed by Ethereum divides transactions into sub-transactions and executes each of them within a designated shard. Pyramid [43] introduces a system comprising i-shards, dedicated to Single-shard TX*s*, and b-shards, designed for Cross-shard TX*s*. Following the completion of Cross-shard TX*s* processing in the b-shards, a cross-shard consensus is necessary to validate transactions within the corresponding i-shards. Nonetheless, these protocols may encounter challenges associated with high transaction conflicts, as each shard processes Single-shard TX*s* independently without a coordinating entity.

2.2 DAG-based BFT consensus

DAG-based BFT consensus protocols are designed to decouple network communication from the consensus process. These protocols, such as Narwhal [19, 75], BBCA-Chain [52], Shoal/Shoal++ [9, 74], Mysticeti [11], Cordial Miners [47], and Motorway [28], operate by proposing blocks in rounds. Each block contains a collection of transactions and references to previous blocks. These interconnected blocks form a DAG, where the blocks represent vertices and the references between them represent edges. The causal history

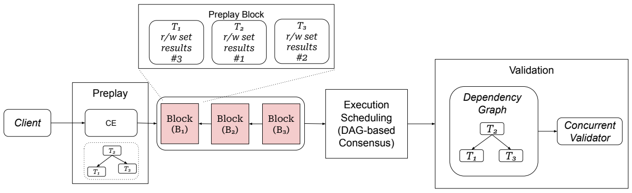

Figure 1: Single-shard TX*s* in each shard will be executed by the *CE* to obtain their execution outcomes. Then, the blocks with the execution outcomes will obtain a global order through a DAG-based consensus protocol and will be validated by each replica. The validation will generate a dependency graph based on the read/write sets in the blocks to improve the parallelism.

of a block B refers to the sub-graph that starts from B. When processing the block B_r in round r, blocks from previous rounds in the causal history of B_r are also processed implicitly.

As each edge in the DAG represents a vote, the DAG also functions as a consensus protocol, enabling each node to establish the total order of all blocks in the DAG without the need for communication with other nodes. The consensus protocol ensures that all nodes receive the same order of committed blocks.

DAG-based protocols provide the following properties:

- Validity: if an honest replica R has a block B in its local view of the DAG, then R also has all the causal history of B.
- Block Consistency: if an honest replica R obtains a block B_r in round r from replica P, then, eventually, all other honest replicas will have B_r.
- Completeness: if two honest replicas have a block B_r in round r, then the causal histories of B_r are identical in both replicas.

3 Thunderbolt Overview

Thunderbolt is designed to enhance the efficiency of smart contract execution through sharding and provides a shard reconfiguration to mitigate censorship attacks, such as dropping blocks post-execution or avoiding proposing selected transactions.

Like traditional sharding systems, Thunderbolt distributes transactions across various shards. However, Thunderbolt does not separate the replicas into different shards where each shard runs individual consensus protocols. They instead jointly run a consensus protocol. Each replica in Thunderbolt acts as a shard submitter and processes the transactions in its shard. To simplify the illustrations, we may interchange replicas and shard submitters where evident. Then, Thunderbolt leverages a DAG-based protocol to run a consensus to agree on the execution among all the shards.

Thunderbolt applies the *EOV* model to execute the Single-shard TX*s* in a non-deterministic order by a concurrent executor (*CE*) from each shard submitter and validate the results among all the shards after the ordering. It then employs the *OE* model to execute the Cross-shard TX*s* through a deterministic optimistic concurrent control protocol after the consensus to provide an atomic commitment.

Thunderbolt also allows the migration of each shard to another replica to avoid a censorship attack, such as dropping blocks or avoiding proposing selected transactions.

3.1 System Model, Goals, and Assumptions

In this section, we describe the system model and design goal. We leave the discussion of the Single-shard TX*s* in Section 4, the Cross-shard TX*s* in Section 5, and the shard reconfiguration in Section 6.

Threat model. We assume a set of n replicas, of which at most f are faulty, $n = 3f+1$. The f faulty replicas can perform any arbitrary (Byzantine) failures, while the remaining replicas are assumed to be honest and follow the protocol's specifications at all times. We assume an eventually synchronous network [22] that messages sent from a replica will ultimately arrive in a global stabilization time (*GST*), which is unknown to the replicas. We also assume communications between replicas go through authenticated point-to-point channels, and messages are authenticated by a public-private key pair signed by the sender.

Design goals. We guarantee basic serializability, safety, and liveness. Intuitively, serializability means that execution produces the same result as a sequential execution across all the replicas. Safety means that every correct node receiving the same sequence of transactions performs the same state transitions. Liveness means that all correct nodes receive the same set of transactions and eventually execute them.

Definition 1 (Seriazability). An honest replica holds the same validation outcomes when executing the same block of transactions.

Definition 2 (Safety). All honest replicas agree on the same block of transactions in each round.

Definition 3 (Liveness). Each honest replica will eventually decide a block of transactions.

Data model. The data model assumes that each transaction includes a contract code with functions to access data belonging to the sender in the shard. The contract involves two types of operations: $<Read, K>$ and $<Write, K, V>$. Here, K represents the key required for access, and V is the value that needs to be written to the key K. The contract code is Turing-complete and users could not obtain any information without execution.

Thunderbolt groups users and their data into distinct shards. To distribute the data into different shards, users must provide the shard ID *SID* for each key to indicate where to access the keys. Thus, a transaction is a Single-shard TX if it contains one *SID*, otherwise, it is a Cross-shard TX.

4 Single Shard Transactions

Thunderbolt comprises three primary components to process Single-shard TX*s*: preplay, execution scheduling, and validation. During each round, a shard submitter executes a batch of the Single-shard TX*s*, generating a block containing the execution outcomes. Then, the submitter transmits the block to other shards via a DAG-based consensus protocol. The validation of the block occurs in parallel across other shards during the consensus. Once a replica commits a block, the replica will record the outcomes of each transaction inside the block. Figure 1 shows the data flow of processing Single-shard TX*s*.

4.1 Preplay

In Thunderbolt, each shard submitter R is responsible for executing transactions and obtaining execution outcomes while generating a block B_r before disseminating it through the DAG in round r.

R runs a concurrent executor (CE) to execute transactions in batches and generates detailed outputs for each transaction. These outputs include read/write sets, scheduled order, and operation results. The scheduled order provides the serialized execution order that can obtain the operation results, while the read/write sets give the keys that each transaction accessed.

4.2 Execution Scheduling

Thunderbolt supports any DAG-based data dissemination layer equipped with a consensus protocol (section 2.2) to determine the total order of blocks among replicas. In each round r, R delivers B_r to the DAG to generate a node in the graph that contains edges to all the blocks in previous rounds, including the ones that R proposed in round $r - 1$. During the consensus, each replica will validate the results included in the block. It's essential to note that from the completeness property (section 2.2), block B_{r-1} in round r should be validated before block B_r from the same shard since block B_r will have a link to the block B_{r-1} from the same shard to the execution.

4.3 Validation

Upon receiving block B_r of round r through the DAG, Thunderbolt will initiate the verification process to ensure the execution results of each transaction within the block are correct. This process is accomplished via the read/write sets to construct a dependency graph locally. The graph serves to allow the validators to process transactions in parallel instead of sequentially validating them, thereby improving the overall performance.

To verify these transactions, Thunderbolt leverages the native OCC protocol [49], and applies to any other deterministic protocols, which executes a set of validators to verify the transactions in parallel in deterministic orders.

A valid dependency graph provided by the shard submitters consistently generates the same results on the read sets and the final values written on each key are the same as the ones recorded in the block. Thus, if the validation produces a mismatch in the values from the read sets, Thunderbolt will disregard the block.

5 Cross-shard transactions

Cross-shard TX*s* are transactions that involve multiple shards. Related work utilizes 2PC-based protocols to lock data in the relevant shards and maintain the lock until the transactions are committed. However, this approach requires one to lock the whole shards when the transaction input set is unknown, which leads to performance degradation.

In response to this challenge, Thunderbolt proposes an alternative approach to handle Cross-shard TX*s* by leveraging the leaders of the underlying DAG to execute transactions in deterministic order.

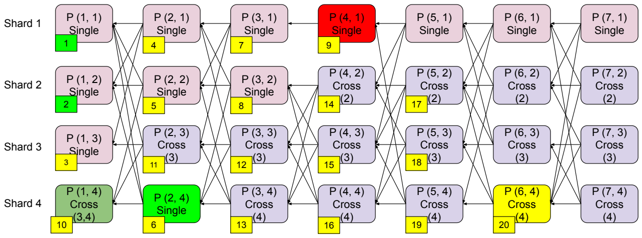

Figure 2: An example illustrates the sequence of the execution order across Cross-shard TX*s*, depicted within the boxes. The color of the boxes indicates which of the leaders commits that transaction. $P(i, j)$ denotes the transaction proposed by the shard submitter of shard j in round i. Shard 4 is the leader in round 2 and round 6 ($P(2, 4)$ and $P(6, 4)$). In this example, the absence of a reference from the first leader $P(2, 4)$ to $P(1, 3)$, which is essential for the Cross-shard TX $P(1, 4)$, results in the non-commitment of both $P(1, 4)$ and $P(2, 4)$. Furthermore, the delay from the second leader $P(4, 1)$ at shard 2 causes the shard submitter to convert the Single-shard TX $P(4, 2)$ into a Cross-shard TX, consequently postponing its execution. Finally, in round 6, the leader $P(6, 4)$ commits all the transactions in its history (with yellow boxes) and ensures the commitment of all Single-shard TX*s*, including $P(2, 4)$, before executing the Cross-shard TX*s*.

When a proposer initiates a Cross-shard TX, Thunderbolt will directly route the transaction to the DAG without undergoing the preplay and one of the leaders will execute the Cross-shard TX in a determined order. Consequently, in the unlikely scenario where all transactions are Cross-shard TX*s*, Thunderbolt will function as a standard system that employs the *OE* model.

Since Thunderbolt executes Single-shard TX*s* in the preplay phase (Section 4.1), Thunderbolt must ensure a consistent partial order between Single-shard TX*s* and Cross-shard TX*s* within each shard by adhering to the following rules:

G1) When a leader L commits both a Single-shard TX and a Cross-shard TX, the Single-shard TX must be committed before the Cross-shard TX.

G2) If a leader L_i commits a Cross-shard TX X in round i and a Single-shard TX Y will be committed by another leader L_j in round j where $j > i$, Y cannot be executed until X is committed.

To accomplish this goal, we made modifications to the DAG protocol:

P1) When a shard submitter proposes a Cross-shard TX X, the shard submitter directly delivers X to the DAG, bypassing the *CE*.

P2) When a shard submitter SL proposes a Single-shard TX X in round r and the DAG selects a leader L in the current round and $SL \neq L$, SL must wait for the proposal from L before preplaying X. If there is any uncommitted Cross-shard TX Y that conflicts with X in the history of L, SL will convert X into a Cross-shard TX and convey it to the DAG. Otherwise, SL will preplay X and deliver its results.

P3) When a leader L commits the proposal comprising a series of transactions, L will execute all the Single-shard TXs before the Cross-shard TXs.

P4) When a leader L in round r commits a Cross-shard TX X in its history that related to shard A but L does not include the proposal from A in round $r - 1$, L will skip to commit the proposals from A after X.

The modified protocol improves Thunderbolt to handle Cross-shard TXs without impeding any shards, while also allowing the execution of Single-shard TXs during the preplay phase to achieve greater parallel execution. Figure 2 provides an illustrative example involving Single-shard TXs and Cross-shard TXs. Rules P3 and P4 guarantee that before a leader finalizes a Cross-shard TX, L has executed all conflicting Single-shard TXs.

5.1 Parallel Execution

During executing the Cross-shard TXs, Thunderbolt retains all sharding information for each transaction. Instead of processing transactions sequentially, Thunderbolt can employ a deterministic concurrency control technique, such as QueCC [60], to construct a dependency graph based on their cross-shard information, thereby enhancing parallelism.

5.2 Message Failures

Unfortunately, network issues will delay the messages. If the leader L is unable to include all the Single-shard TXs related to a Cross-shard TXs due to network issues, L will bypass the Cross-shard TX and its following transactions from the same shard. This is because including incomplete transactions would violate the rule G2. Then, these transactions will be committed by later leaders. However, to prevent a Cross-shard TX from blocking all future transactions in the same shard due to the crossing shard failure, the Cross-shard TX will be dropped if it cannot be committed in limited rounds. Similarly, if a shard submitter does not receive a proposal from the leader within a specific time limit during a round, it cannot prepay its Single-shard TXs. Then the shard submitter will deliver the Single-shard TX to the DAG as a Cross-shard TX, such as the $P(4, 2)$ in Figure 2.

5.3 Preplay Recovering

Rule P2 indicates that if a shard submitter SL which proposes a Single-shard TX identifies a conflict Cross-shard TX in the leader's history, SL will transition the Single-shard TX to a Cross-shard TX, like $P(2, 3)$ in Figure 2. However, this transition results in the inability to preplay the subsequent Single-shard TXs, leading to a loss of the benefit gained from the preplay.

To retrieve the replay of a Single-shard TX, it is essential for SL to ensure that all the conflict Cross-shard TXs have been executed by the preceding leaders. To determine if the leader CL_r in round r is eligible to commit, SL needs to skip proposing Single-shard TXs for two rounds via proposing empty proposals if it is not the leader in round r. However, SL can still propose Cross-shard TXs in these rounds, like $P(4, 3)$ in round 4 in Figure 3. Subsequently, if CL_r is a valid leader, SL is allowed to preplay the Single-shard TX after committing the proposals from CL_r and delivering its outcomes to the DAG. Otherwise, SL will persist in transitioning the Single-shard TX to a Cross-shard TX. If SL is the leader in round r, it can preplay the Single-shard TX in round $r+1$ directly after committing the proposals in round r.

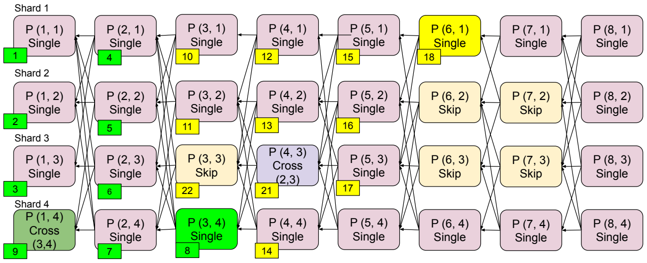

Figure 3: An example of skipping two rounds to validate the leaders and restart the preplay of the Single-shard TXs. The shard submitter of shard 3 identifies the occurrence of $P(1, 4)$ in the history of the leader $P(3, 4)$ in round 3. Subsequently, the shard submitter stops proposing the Single-shard TX in round 3, but still proposes a Cross-shard TX in round 4. After that, it becomes viable to recommence the preplay of the Single-shard TX in round 5 and the future rounds. Later, given that $P(4, 3)$ in round 4 is also a Cross-shard TX, it is imperative for the leaders of shard 2 and shard 3 to abstain from proposing a Single-shard TX until round 8.

Additionally, this strategy requires a minimum three-round period to select leaders. Like the leaders in round 3 and round 6 ($P(3, 4)$ and $P(6, 1)$) in Figure 3.

6 Shards Reconfiguration

In a Byzantine environment, malicious attacks may compromise the security and integrity of a replica. Once a replica falls under the control of malicious actors, the transactions within the assigned shard may become susceptible to censorship attacks.

Thunderbolt employs a round-robin selection mechanism [70] to rotate the shard submitters when a leader fails to propose transactions for K rounds or at intervals of K' rounds, where $K' > K$. This is also the key technique enabling Thunderbolt to prevent malicious clients from submitting the same transactions to all replicas of the system and degrade its throughputs that each shard submitter can locally perform transactions deduplication to prevent the same transaction from being proposed multiple times, which is a key open challenge of DAG-based protocols [10, 14, 75].

Diverging from traditional consensus protocols that depend on notification messages to alter primary nodes, Thunderbolt introduces an innovative mechanism that leverages the underlying DAG protocols to facilitate the seamless transition to a new DAG and reconfigure the shard submitters. We leverage a round-robin approach to select a new submitter that if the current submitter of shard X is replica R_i, the subsequent submitter of shard X will be $R_{(i \ mod \ n)+1}$.

However, the transmission of blocks to a new submitter may experience delays or omissions due to network issues or the actions of a malicious submitter. If the new submitter for round r is unable to receive the proposal committed in round $r - 1$ from the previous

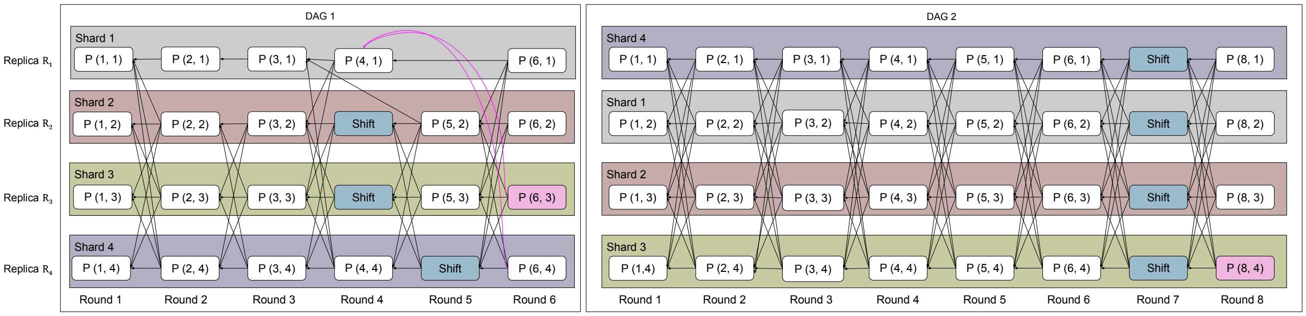

Figure 4: Each replica will propose a Shift block at round r if the block from a shard submitter from round $r-2$ ($K=2$) does not arrive or has received 2 Shift blocks at round $r-1$. Thunderbolt triggers a shard reconfiguration when a leader commits a block (pink block at round 6) including Shift blocks from 3 replicas. Then Thunderbolt generates a new DAG (DAG 2). All the results from the uncommitted blocks, like $P(6,1)$, will be discarded before starting the transactions in DAG 2. In DAG 2, although every replica proposes blocks in every round, Shift blocks will still be sent after $K^{'}=6$ blocks have been proposed to rotate the shard submitters to avoid censorship attacks.

submitter, the new submitter will halt the operations until the block arrives to ensure safety.

To address this challenge, Thunderbolt introduces Shift blocks to reach agreements among shards when a shard reconfiguration should be initiated and switch to a new DAG to process further transactions.

A replica R broadcasts a Shift block in round r under the following conditions:

(1) R receives no block from a shard submitter after round $r-K$.
(2) R has proposed blocks for at least $K^{'}$ rounds.
(3) R received $f+1$ Shift blocks from distinct replicas at round $r-1$.
(4) R does not broadcast the Shift block before.

In the scenario shown in Figure 4 where $K=2$ and $K^{'}=6$, a replica triggers the broadcasting of a Shift block. During round 4, replica R_2 and R_3 do not receive any blocks in rounds 2 and 3. Subsequently, R_2 and R_3 broadcast a Shift block to other replicas. During round 5, despite replica R_4 having received blocks at round 4 from replica R_1, it still broadcasts the Shift block because it has received 2 Shift blocks at round 4 to ensure the liveness.

Since all the honest replicas will commit the same block at the same round (section 2.2), we mark the round of the first commit block that includes $2f+1$ Shift blocks to be the ending round for the current DAG. Then, **each replica will start the new DAG at the same ending round to guarantee the safety of the system**. All the transactions not committed in the ending round will be discarded and re-executed in the future DAG. For example, R_2 will propose $P(5,2)$ at round 5 after proposing a Shift block at round 4. Finally, the block $P(6,3)$ from R_3 at round 6 is selected as the leader during the consensus. R_3 commits all the history, including the Shift blocks from other replicas. Then, all the replicas will switch to the new DAG (DAG 2) and start executing the transactions within the new shard. Furthermore, each replica propose a Shift block in $K^{'}=6$ blocks in the new DAG (DAG 2) to trigger a transition to the next DAG to safeguard against censorship attacks.

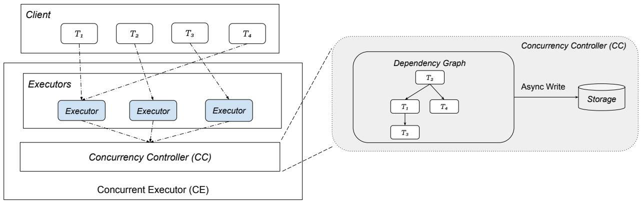

Figure 5: The architecture of Concurrent Executor. A set of executors executes transactions and the concurrency controller uses a dependency graph to determine the order of the transactions and the execution results.

7 Concurrent Executor

The concurrent executor (*CE*) is a crucial component that enables Thunderbolt to process Single-shard TX*s* concurrently in the preplay phase. *CE* outputs a sequential order, read/write sets, and execution results for a batch of transactions that can be verified by any other replica. The sequential order produced by *CE* can be any order different from the arrival order of the transactions. Thus *CE* is a nondeterministic concurrency control executor.

The architecture of *CE* is illustrated in Figure 5, where a set of executors executes transactions, and a Concurrency Controller (*CC*) determines the execution results among the transactions. Transactions undergo a two-phase data flow process, which involves an execution phase and a finalization phase.

7.1 Execution Phase

During the execution phase, the executors access the data within *CC* directly and *CC* maintains a dependency graph to keep track of the relationship between transactions and all the results are stored in the graph directly to avoid accessing the disk IO. The critical characteristic of *CC* is *CC* only maintains the graph based on the current operations among the transactions without requiring any read/write sets knowledge. Furthermore, *CC* is non-deterministic and can arrange transactions in any order. For example, if two transactions T_1 and T_2 update on the same keys, a dependency

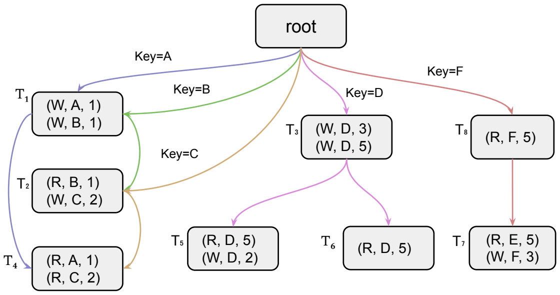

Figure 6: Dependency Graph on *Thunderbolt*. Edges with the same color represent a dependency graph with a specific key.

edge between T_1 and T_2 may not be created since T_1 and T_2 can be arranged in any order in the current state.

While receiving an operation from a transaction T sent by the executors identified accessing the key K, denoted as O_k, *CC* checks the relationships among the transactions. If T conflicts with other transactions or has been aborted by other transactions, the operation O_k will not be considered valid. The transaction T will be aborted in such cases and require re-execution. Otherwise, if the operation O_k is valid, it will be added to the dependency graph (section 8.1) and obtain the operation result, such as the value V that O_k intends to read. We can obtain the operation results from other transactions directly based on the dependency graph to allow for reading uncommitted data.

7.2 Finalization Phase

During the finalization phase, the executor informs *CC* that the executor has completed all the operations. Then *CC* will update the results to the storage asynchronously once all its dependencies have been committed and assign the execution order to the transactions. If *CC* has terminated the transaction due to conflicts with other transactions, *CC* aborts the transaction and notifies the executor to restart the execution.

8 Dependency Graph in CC

This section describes the dependency graph G at the heart of the *CC* component, which plays a crucial role in maintaining the causal relationship between transactions during the replay in *CE*. *CC* ensures that the sequential order of the execution generated by G is a 'valid' order.

8.1 Dependency Graph Construction

A Dependency Graph is a graph $G(V, E)$ that plays a crucial role in tracking the causal relationship between transactions in *CC*. Each node $v \in V$ represents a specific transaction. Additionally, each edge $e(u, v, k) \in E$ indicates a connection between two transactions u and v on a key K. This relationship is represented as $u \longrightarrow_k v$. For example, in Figure 6, transaction T_5 generates an edge $e(T_3, T_5, D)$ from T_3 because T_5 acquires the value 3 of key D from T_3.

Without loss of generality, we have assigned a root node denoted as R and added edges $e(R, u, k) \in E$ for each $u \in V$ that accesses key K but does not have any incoming edge on key K, such as T_7 and T_8 in Figure 6.

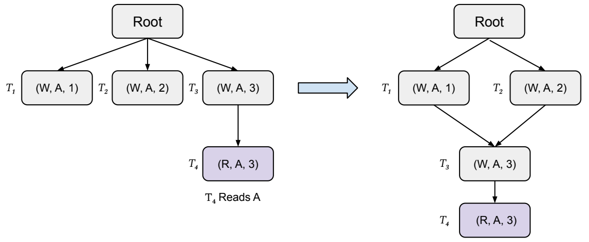

Figure 7: An example to add a new read operation. Relationships between the nodes are modified to guarantee correctness.

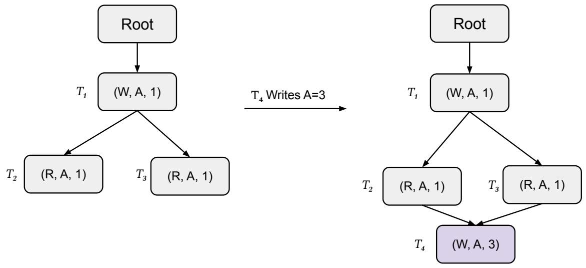

Figure 8: An example to add a new write operation that adds dependencies from all the read nodes.

If the graph G is acyclic, then a sequential order can be established by generating a topological order from it. As outlined in section 3.1, it is crucial that every transaction must obtain the same causal order in any topological order from G to ensure consistency. Therefore, G is considered a valid graph only if any sequential order generated from the topological order is a valid serialization order and produces the same outcomes. By following any correct order, all transactions will yield the same execution results. However, due to the non-deterministic characteristic, the results may not be the same as the one executed in their arrival order.

Each node u maintains all the records of the operations triggered by a transaction u, including the resulting values. Since a transaction is an atomic commitment, we combine the internal operations to simplify the in-node states. However, to trace the conflicts between two nodes, we must retain the first operation if it is a read and the last operation if it is a write, to ensure that the causal relationship is not lost. Thus, we remain at most two operations in the nodes: the first read and the last write.

To help illustrate the algorithm, we define the types of each node depending on the operations it contains on a key:

- A node $v \in V$ is a read node R_v^k if the first operation on key K is a read.
- A node $v \in V$ is a write node W_v^k if v contains write operations on key K.
- The root node R is a write node.

8.2 Generating New Nodes

This section presents the process of adding operations from a new transaction to the dependency graph G.

CC creates a new node whenever an operation O_k is received from a new transaction T. If O_k is a write operation, T needs to

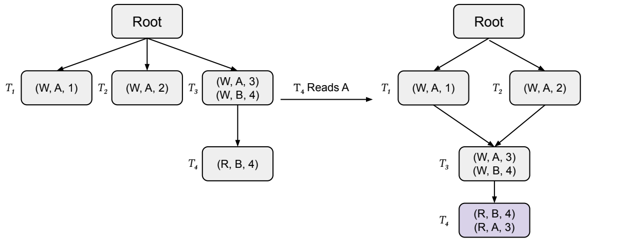

Figure 9: T_4 reads key A on its existing node and obtains the value from T_3.

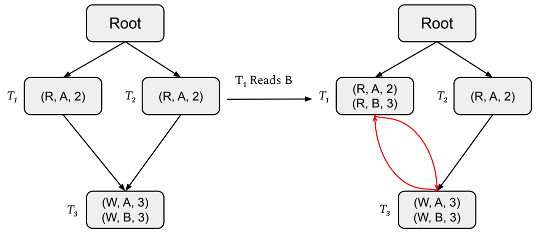
Root
T_1 (R, A, 2)
T_2 (R, A, 2)
T_3 (W, A, 3) (W, B, 3)
T_1 Reads B
Root
T_1 (R, A, 2) (R, B, 3)
T_2 (R, A, 2)
T_3 (W, A, 3) (W, B, 3)

Figure 10: T_1 reads B and adds a dependency from T_3 following the rules in Section 8.2, which results in a cycle of conflict.

establish a connection to each casual relation. To avoid pointing to the root and assuming that the earlier transaction will commit first, the non-write nodes v on key K, which only contains reads, without any outgoing edges (not dependent by other nodes) are selected, and edges $e(v, u, k)$ are added, pointing to u (Figure 8).

On the other hand, when a read operation is performed, *CC* selects the latest write node u in order to obtain the latest value, or selects the root to read the data value from storage if no write nodes exist. If the write node u is selected, we need to make all other write nodes W_v^k contain a path to u to guarantee the correctness for the read after write between u and v. Finally, the operation and its result <Type, Key, Result> will be written into the node u. An example of adding a new read operation on A from T_4 is depicted in Figure 7. T_4 selects T_3 to read to obtain $A = 3$ and adds an edge from T_3 and a record <R, A, 3> is logged down in the node. Then T_3 will also add two edges from T_1 and T_2 respectively.

8.3 Operations on Existing Nodes

When receiving an operation O_k for key K from an existing transaction T in G, *CC* will select the corresponding node u to append the record. If O_k is a read operation, the result will be directly retrieved if u contains the record for key K. Otherwise, it will proceed with the new node operation as specified in section 8.2 to choose a previous one to access the value. Figure 9 illustrates an instance where T_4 reads key A as its second operation and retrieves the value from T_3. If O_k is a write operation, the operation will be appended to the node.

8.4 Conflict Detection

Appending the records to an existing node may lead to transaction conflicts, like a transaction updates the value again but it has been read by another transactions or a dependency cycle is created due to the dependency on another key since we always find the latest write to retrieve the value. Figure 10 depicts a scenario in which T_1 attempts to retrieve the value of B from T_3, which has established a dependency from T_1 due to key A, thereby resulting in the creation of a dependency cycle. In this case, *CC* will try to read the value from its ancestor, like B reads the value from the *Root* in Figure 10. If there is still any conflict with other transactions, *CC* will trigger the abort process.

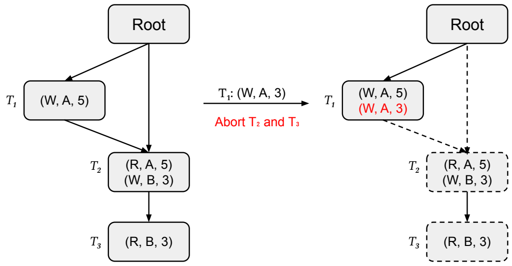

Figure 11: Cascading aborts from T_1 since T_1 wants to write A that breaks the read on T_2.

Once conflicts are detected, *CC* triggers an abort process as follows:

(1) If u only contains read operations, abort T itself.
(2) If u contains write operations, cascading abort from T.

In Figure 11, we need to abort T_2 and T_3 since T_1 contains a write operation. However, in Figure 10, we only need to remove T_1 and keep T_3 alive.

8.5 Asynchronous Commit

Each node in G stores the outcomes of all the operations from a transaction, enabling quick and easy access to value with a specific key from the incoming node during a read operation. We define the incoming node as follows:

Definition 4. An incoming node of a read node u on key K is defined as the node v, which contains records on the given key K and has a dependency path to u. If node v is both an incoming node and a read node on key K of u, it is an incoming read node on key K of u. Similarly, if node v is both an incoming node and a write node on key K of u, it is an incoming write node on key K of u.

Definition 5. An direct incoming node of a read node u on key K is defined as the node v, which is an incoming node of u on key K and there is no other incoming node w on the dependency path from v to u.

As a result, all operations can be executed in memory, in which each read operation can obtain the result from its direct incoming node, resulting in a transaction that can be committed asynchronously without incurring high latency from blocking other transactions.

9 Evaluation

This section presents an evaluation of Thunderbolt by measuring the performance on *CE* and the Thunderbolt framework. We implement Thunderbolt on Apache ResilientDB (Incubating) [1, 36]. Firstly, we will compare *CE* to two baseline protocols: OCC [49] and 2PL-No-Wait [72]. Additionally, we will study the performance of Thunderbolt built on Tusk [19] and also use Tusk as our baseline.

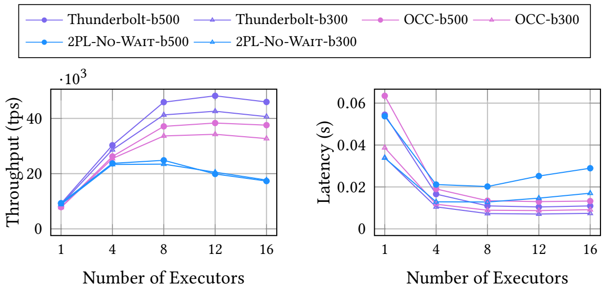

Figure 12: Throughput and latency of different numbers of executors with $P_r = 0.5$.

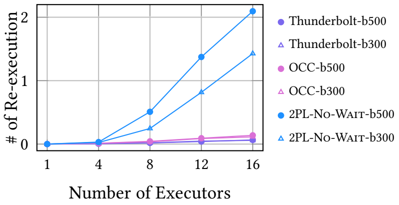

Figure 13: Average retry time of different numbers of executors with $P_r = 0.5$.

We use SmallBank [2] as the input workload, a benchmarking suite that simulates common asset transfer transactions.

9.1 Baseline Protocols

We implement OCC [49] and 2PL-No-Wait [85, 86] to compare the performance against our concurrent executor. We set up our experiments on AWS c5.9xlarge consisting of 36 vCPU, 72GB of DDR3 memory. We use LevelDB as the storage to save the balance of each account.

9.2 Experiment Setup With Smallbank

SmallBank [2] is a transactional system that comprises six distinct transaction types, five of which are designed to update account balances, while the remaining transaction is a read-only query that retrieves both checking and saving the account details of a user. Our focus is on two types of transations: SendPayment and GetBalance, which are used to transfer funds between two accounts and retrieve account balances, respectively. Our objective is to evaluate the performance under varying read-write balance workloads. During a SendPayment transaction, account balances are updated by reading the current balance and then writing the new values back. We have created 10,000 accounts and conducted each experiment 50 times to obtain the average outputs.

We evaluated the impact of parallel execution. We measured the performance by uniformly selecting GetBalance with a probability of P_r while SendPayment with $1 - P_r$. To select accounts as transaction parameters, we followed a Zipfian distribution and set the Zipfian parameter θ. The value of θ determines the level of account contention, with higher values leading to greater contention. We only focus on the data workloads with high contention by setting $\theta = 0.85$.

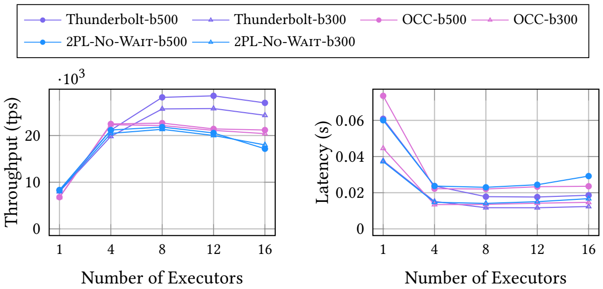

Figure 14: Evaluation of different numbers of executors with update only workload ($P_r = 0$).

9.3 Impact from Concurrent Executor

We first evaluate the impact of increasing the number of executors to execute the transactions then measure the aborts produced by each protocol. We ran two batch sizes $b300$ and $b500$ for each protocol: Thunderbolt-b300, Thunderbolt-b500, OCC-b300, OCC-b500, 2PL-No-Wait-b300, and 2PL-No-Wait-b500. We set $P_r = 0.5$ to measure a read-write balanced workflow and $P_r = 0$ on an update-only workflow.

Number of Executors. In the read-write balanced workflow, the results depicted in Figure 12 show that 2PL-No-Wait protocols with different batch sizes all experience a drop in performance when increasing the number of executors beyond 8. However, Thunderbolt and OCC protocols with all the batch sizes obtain their highest throughputs on 12 executors and maintain stable throughput. Thunderbolt-b500 obtained $43K$ TPS while OCC-b500 achieved $35K$ TPS.

In the update-only workflow, the results shown in Figure 14 indicate that OCC and 2PL-No-Wait stopped increasing earlier at 4 executors (both around $22K$ TPS) while Thunderbolt provides a peek throughput ($28K$ TPS) on 12 executors.

These experiments demonstrate that all the protocols do not obtain significant benefits for a large number of executors in a high-competition workflow. However, Thunderbolt still can achieve more parallelism with more executors.

Evaluation of Abort Rates. As we increased the number of executors, we also measured the average number of re-execution for the transactions. The results in Figure 13 indicate that when the number of executors goes beyond 8, all 2PL-No-Wait protocols experience a significant increase in the rate of abortions, leading to a drop in throughput from $24k$ to $18k$ in the read-write balanced workflow. While OCC protocols provide a lower rate within the read-wirte balanced workflow. However, Thunderbolt achieves the lowest abortions, with Thunderbolt-b500 reducing 50% of the abortions from OCC-b500 and 90% from 2PL-No-Wait-b500 in all the experiments.

9.4 System Evaluation

We conducted evaluations to determine the impact of Thunderbolt built on Tusk. In our evaluation, we compared the performance of Thunderbolt with Tusk, which executes transactions in order after reaching a total order after DAG protocols. We also leveraged SmallBank as the input workload.

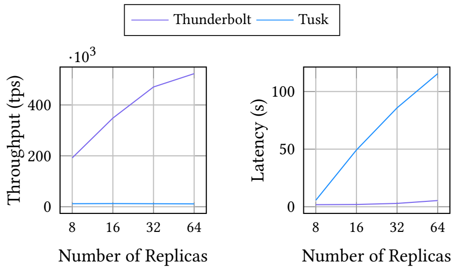

Figure 15: Throughput and average latency within different replicas.

For each replica, we set up a *CE* with 16 executors to execute the transactions with a batch of 500 and 16 validators to validate the block after consensus. We scaled the system from 8 replicas to 64. By default, we have set K' to a large value to prevent rotation. At the end of our evaluation, we will assess the effects of various K' which will trigger the shard reconfiguration(§9.4).

SmallBank. Within the experiments with Smallbank workload, we only focus on the read-write balanced scenario ($P_r = 0.5$) that half of the transactions are read-only. The addresses in the transactions are selected in 1000 users with $\theta = 0.85$ to simulate a high contention workload. The results in Figure 15 show that Thunderbolt achieved higher throughput than the sequential order in Tusk which speeds up 50X that Thunderbolt obtained 500K TPS while sequential execution only obtained 11K TPS. The results also demonstrate while increasing the replicas, the throughput increased when executing the transactions before the consensus, which reduces the bottleneck from the sequential execution.

Cross-shard Transactions. Then, we conducted an examination of the impact of Cross-shard TXs within 16 replicas. We randomly designated the percentage P% ($0 < P \leq 100$) of the transactions to be processed by two shards. Furthermore, we tested the impact of parallel execution by comparing the sequential execution, denoted as Thunderbolt *, on the Cross-shard TXs.

Figure 16 reveals that as the percentage P increases, the performance of both Thunderbolt and Thunderbolt * decreases. In scenarios involving only single-shard transactions ($P = 0$), both Thunderbolt and Thunderbolt * achieved 100K TPS. However, with an increase in the percentage P to 8%, Thunderbolt * experienced a drop to 16K TPS, while Thunderbolt achieved a higher TPS (64K). Additionally, Thunderbolt * demonstrated a throughput similar to Tusk, approximately 10K TPS. Nevertheless, Thunderbolt outperformed by delivering 19K TPS when all transactions were *CSTs* due to their parallel execution.

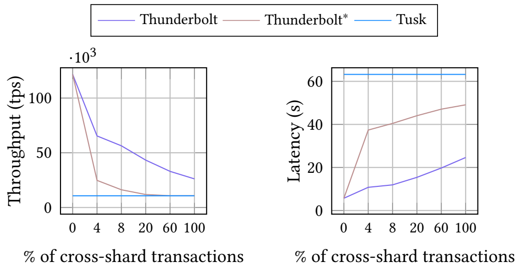

Figure 16: Throughput and average latency within different ratios of cross-shard transactions within 16 replicas.

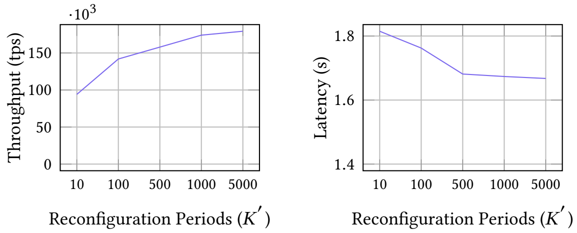

Figure 17: Throughput and average latency within different reconfiguration periods within 8 replicas.

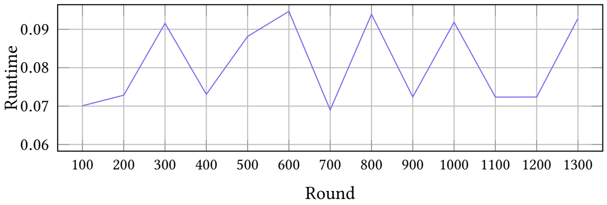

Figure 18: Average latency per 100 rounds and reconfigure the shard per 300 rounds.

Reconfiguration Periods. Now, we analyzed the performance using different reconfiguration periods K' to transition the shard submitters into a new DAG on 8 replicas. Figure 17 demonstrates that Thunderbolt exhibited lower performance with smaller K' values (80K TPS with $K' = 10$), attributed to the costly transition between DAGs. Conversely, when K' was increased to over 1000, Thunderbolt demonstrated significantly improved stability, achieving a throughput of 180K TPS. Additionally, the average latency decreased as K' increased from 1.9s to 1.7s. Figure 18 also shows the average run time of committing proposals per 100 rounds, that is $\frac{1}{100}\sum(T_{commit(i)} - T_{commit(i-1)})$ where $T_{commit(i)}$ is the time of committing round i. We set K' as 300 and it demonstrated Thunderbolt will not get stuck during the reconfiguration. The runtime of each round is around 0.07s to 0.1s.

Failures. Finally, we evaluated the impact of replica failures within 16 replicas. We forced f replicas ($f = 1$ or $f = 2$) to stop working during the experiments. We randomly designated a percentage P% ($0 < P \leq 100$) of the transactions to be processed by two shards.

Figure 19 reveals that Thunderbolt still can provide higher throughputs when some shards stop working. When one replia failed to propose transactions, Thunderbolt
1 ($f = 1$) obtained 78K TPS working on all Single-shard TXs ($P = 0$) and 17K TPS on all Cross-shard TXs while when two replicas failed to propose transactions, Thunderbolt
2 ($f = 2$) obtained 66K TPS on all Single-shard TXs and 15K TPS on all Cross-shard TXs.

10 Related work

We discuss other prior works relevant to Thunderbolt and the comparison.

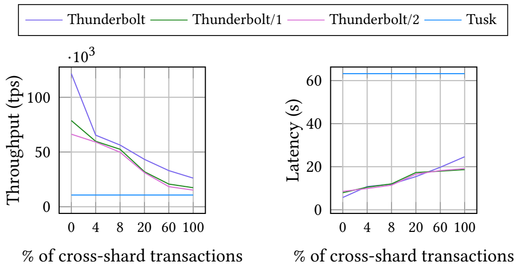

Figure 19: Throughput and average latency within different ratios of cross-shard transactions within 16 replicas when f ($f = 1$ or $f = 2$) replicas failed.

10.1 Sharding

Numerous studies [4, 20, 37, 41, 48, 62, 80, 88] has been carried out on the necessity of sharding to improve the scalability in blockchain systems.

Omiledger [48], Dang [20], and RepidChain [88] propose the provision of shards by generating committees with random coins. However, these existing protocols rely on a random assignment based on a distributed random coin generation on the committees and re-assign the committee members in each epoch.

In contrast, Thunderbolt takes a different approach by making each replica as a shard leader and leveraging the DAG to live migration to a new DAG to rotate the leader if the replicas detect malicious attacks. The assignment and reconfiguration in Thunderbolt are much more efficient than previous work.

Execute-Order-Validate. The Execute-Order-Validate (EOX) framework was originally introduced by Hyperledger Fabric [8] offering the advantage of allowing transactions to be optimistically executed by a subset of executors before obtaining a global order. However, the parallel execution may lead to conflicts between two transactions, causing the validation to abort the transactions after being ordered.

Various techniques have been proposed to improve the bottleneck on Hyperledger, including optimizing the peer process, replacing the state DB with a hash table, and pipelining the execution [30, 78, 79] Other platforms like Fabric++ [67], XOXFabric [29], and FabricSharp [65] have also been developed to reorder transactions within a block by analyzing their dependencies after the consensus, which reduces the abort rate.

Thunderbolt, inspired by Hyperledger, has implemented an execution model that allows transactions to be executed before ordering. However, different from Hyperledger, Thunderbolt distributes transactions into different shards and transactions in each shard will be executed by a shard leader. This approach enables each shard leader to leverage the concurrent executor to execute transactions in parallel while also reordering them to improve performance and reduce the abort rates.

Concurrent Execution. Deterministic approaches have been proposed [24, 60, 84] to make the execution of transactions more efficient. These methods involve constructing a dependency graph to allow transactions to be executed concurrently without causing conflicts. Transaction Chopping [68, 69], SChain [18], and Caracal [61] go one more stop by dividing transactions into smaller pieces before building the dependency graph. Non-volatile main Memory is also introduced in [81] to address long latency transactions.

However, these techniques have some limitations, such as the need to obtain transaction execution read/write sets in advance.

In contrast, Thunderbolt does not rely on the assumption of read/writes, assigns the order dynamically, and minimizes conflicts between transactions.

CHIRON [58] utilizes BlockSTM [27], which provides a non-deterministic execution to extract dependencies (hints) from smart contracts on Ethereum for the acceleration of straggling and full nodes. Block-STM allows smart contracts to be executed in parallel in some order and outputs the read/write sets as results. However, the execution order is based on the arrival time of the transactions.

In contrast, Thunderbolt does not rely on the assumption of arrival time, assigns the order dynamically, and minimizes conflicts between transactions.

10.2 Transaction Reordering

BCC [87] proposed a method to minimize the number of aborted transactions by checking the commit time to adjust the commit order. A directed graph within a batch is constructed in [21] and a greedy algorithm is introduced to reorder transactions to address the NP-hard problem for highly contentious cases. Similar techniques have been applied in [29, 65, 67]. However, these approaches have a higher latency as they aim to find the best global order on a deterministic graph built from the read/write sets obtained from the execution.

In contrast, Thunderbolt builds the graph dynamically and maintains it online. This ensures that the graph is always up-to-date and the order of transactions can be adjusted as needed to achieve the desired results.

10.3 Concurrent Consensus

Blockchain fabric with high performance and scalability is crucial [7, 33, 35, 66]. PoE [34] reduced one phase from Pbft [17] by introducing a speculative execution. RCC [36], FlexiTrust [38], and SpotLess [45] extend the single-leader protocols to multi-leaders to improve parallelism. However, these protocols do not support reconfiguration.

11 Conclusions

We have developed Thunderbolt, a sharding system that applies both Order-Execute and Execute-Order-Validate models to enhance the execution of smart contracts. Thunderbolt distributes transactions into distinct shards and leverages DAG-based protocols to broadcast transactions among all the shards.

Thunderbolt executes single-shard transactions before the consensus while shifting the execution for the cross-shard transactions after the consensus. Thunderbolt leverages the leader underlying the DAG to guarantee the consistency between sing-shard and cross-shard transactions. We also implement a concurrent executor to enhance the execution of single-shard transactions by generating a dependency graph dynamically without any read/write sets known prior.

Thunderbolt also leverages the properties of DAG to migrate the current DAG to a new DAG without a hard stop to rotate the submitters of each shard if a malicious shard submitter is detected.

Our performance evaluation results have demonstrated that Thunderbolt can deliver a 50 times speed-up compared to the native execution provided by Tusk.

12 Acknowledgements

This work is partially funded by NSF Award Number 2245373.

References

[1] [n. d.]. Apache ResilientDB (Incubating). https://resilientdb.incubator.apache.org/

[2] 2019. smallbank benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[3] Rakesh Agrawal, Michael J Carey, and Miron Livny. 1987. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)* 12, 4 (1987), 609–654.

[4] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).

[5] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad Van Moorsel. 2019. OpBench: A CPU performance benchmark for Ethereum smart contract operation code. In *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 274–281.

[6] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1385–1398.

[7] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2022. The Bedrock of BFT: A Unified Platform for BFT Protocol Design and Implementation. *CoRR* abs/2205.04534 (2022). https://doi.org/10.48550/arXiv.2205.04534 arXiv:2205.04534

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

[9] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2024. Shoal++: High Throughput DAG BFT Can Be Fast! *arXiv preprint arXiv:2405.20488* (2024).

[10] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-latency dag consensus with fast commit path. *arXiv preprint arXiv:2310.14821* (2023).

[11] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2023. Mysticeti: Low-Latency DAG Consensus with Fast Commit Path. *arXiv preprint arXiv:2310.14821* (2023).

[12] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[13] Philip A Bernstein and Nathan Goodman. 1984. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 596–615.

[14] Same Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. 2023. Sui lutris: A blockchain combining broadcast and consensus. *arXiv preprint arXiv:2310.18042* (2023).

[15] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.

[16] Michael Casey, Jonah Crane, Gary Gensler, Simon Johnson, and Neha Narula. 2018. The impact of blockchain technology on finance: A catalyst for change. (2018).

[17] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (2002), 398–461. https://doi.org/10.1145/571637.571640

[18] Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Xiaodong Qi, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2021. SChain: a scalable consortium blockchain exploiting intra-and inter-block concurrency. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2799–2802.

[19] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3492321.3519594

[20] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.

[21] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

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

[23] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1597–1609.

[24] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[25] Peter Franaszek and John T Robinson. 1985. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems (TODS)* 10, 1 (1985), 1–28.

[26] Lan Ge, Christopher Brewster, Jacco Spek, Anton Smeenk, Jan Top, Frans Van Diepen, Bob Klaase, Conny Graumans, and Marieke de Ruyter de Wildt. 2017. *Blockchain for agriculture and food: Findings from the pilot study*. Number 2017-112. Wageningen Economic Research.

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

[28] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Motorway: Seamless high speed BFT. *arXiv preprint arXiv:2401.10369* (2024).

[29] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[30] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2020. FastFabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management* 30, 5 (2020), e2099.

[31] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[32] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[33] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. 2023. Chemistry behind Agreement. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org.

[34] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2021. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. In *Proceedings of the 24th International Conference on Extending Database Technology*.

[35] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. *Fault-Tolerant Distributed Transactions on Blockchain*. Morgan & Claypool Publishers. (2021).

[36] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1392–1403. https://doi.org/10.1109/ICDE51399.2021.00124

[37] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (2020), 868–883. https://doi.org/10.14778/3380750.3380757

[38] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2023. Dissecting BFT Consensus: In Trusted Components we Trust!. In *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan (Eds.). ACM, 521–539.

[39] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2020. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 754–764.

[40] Jelle Hellings, Daniel P Hughes, Joshua Primero, and Mohammad Sadoghi. 2020. Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing. *arXiv preprint arXiv:2008.04450* (2020).

[41] Jelle Hellings and Mohammad Sadoghi. 2023. ByShard: sharding in a Byzantine environment. *VLDB J.* 32, 6 (2023), 1343–1367.

[42] Maurice Herlihy. 2019. Blockchains from a distributed computing perspective. *Commun. ACM* 62, 2 (2019), 78–85.

[43] Zicong Hong, Song Guo, Peng Li, and Wuhui Chen. 2021. Pyramid: A layered sharding blockchain system. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 1–10.

[44] Maged N Kamel Boulos, James T Wilson, and Kevin A Clauson. 2018. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. ,

10 pages.

[45] Dakai Kang, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2024. SpotLess: Concurrent Rotational Consensus Made Practical through Rapid View Synchronization. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, Netherlands, May 13-17, 2024*. IEEE.

[46] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 165–175.

[47] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. 2022. Cordial miners: Fast and efficient consensus for every eventuality. *arXiv preprint arXiv:2205.09174* (2022).

[48] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.

[49] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[50] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Ethereum smart contract analysis tools: A systematic review. *IEEE Access* 10 (2022), 57037–57062.

[51] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[52] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2023. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. *arXiv preprint arXiv:2310.06335* (2023).

[53] Microsoft. [n. d.]. eEVM. https://github.com/microsoft/eEVM

[54] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[55] Arvind Narayanan and Jeremy Clark. 2017. Bitcoin's academic pedigree. *Commun. ACM* 60, 12 (2017), 36–45.

[56] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain meets database: Design and implementation of a blockchain relational database. *arXiv preprint arXiv:1903.01919* (2019).

[57] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 124–135.

[58] Ray Neiheiser, Arman Babaei, Giannis Alexopoulos, Marios Kogias, and Eleftherios Kokoris Kogias. 2024. CHIRON: Accelerating Node Synchronization without Security Trade-offs in Distributed Ledgers. *arXiv preprint arXiv:2401.14278* (2024).

[59] Michael Pisa and Matt Juden. 2017. Blockchain and economic development: Hype vs. reality. *Center for global development policy paper* 107, 150 (2017), 1–42.

[60] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[61] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[62] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. 2022. RingBFT: Resilient Consensus over Sharded Ring Topology. In *Proceedings of the 25th International Conference on Extending Database Technology*. OpenProceedings.org, 2:298–2:311. https://doi.org/10.48786/edbt.2022.17

[63] Peter Robinson and Raghavendra Ramesh. 2021. General purpose atomic cross-chain transactions. In *2021 3rd Conference on blockchain research & applications for innovative networks and services (BRAINS)*. IEEE, 61–68.

[64] Peter Robinson, Raghavendra Ramesh, and Sandra Johnson. 2022. Atomic cross-chain transactions for ethereum private sidechains. *Blockchain: Research and Applications* 3, 1 (2022), 100030.

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

[66] Mohammad Sadoghi and Spyros Blanas. 2019. *Transaction Processing on Modern Hardware*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00896ED1V01Y201901DTM058

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

[68] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 325–363.

[69] Dennis Shasha, Eric Simon, and Patrick Valduriez. 1992. Simple rational guidance for chopping up transactions. In *Proceedings of the 1992 ACM SIGMOD International Conference on management of Data*. 298–307.

[70] Madhavapeddi Shreedhar and George Varghese. 1995. Efficient fair queueing using deficit round robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*. 231–242.

[71] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. 2024. Sailfish: Towards Improving Latency of DAG-based BFT. *Cryptology ePrint Archive* (2024).

[72] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial strictness in two-phase locking. In *International Conference on Database Theory*. Springer, 139–147.

[73] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. 2020. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 294–308.

[74] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2023. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058* (2023).

[75] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2718.

[76] Chrysoula Stathakopoulou, Michael Wei, Maofan Yin, Hongbo Zhang, and Dahlia Malkhi. 2023. BBCA-LEDGER: High Throughput Consensus meets Low Latency. *arXiv preprint arXiv:2306.14757* (2023).

[77] Yong Chiang Tay, Nathan Goodman, and Rajan Suri. 1985. Locking performance in centralized databases. *ACM Transactions on Database Systems (TODS)* 10, 4 (1985), 415–462.

[78] Parth Thakkar and Senthilnathan Natarajan. 2021. Scaling blockchains using pipelined execution and sparse peers. In *Proceedings of the ACM Symposium on Cloud Computing*. 489–502.

[79] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*. IEEE, 264–276.

[80] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. 2019. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 41–61.

[81] Yu Chen Wang, Angela Demke Brown, and Ashvin Goel. 2023. Integrating Non-Volatile Main Memory in a Deterministic Database. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 672–686.

[82] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.

[83] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[84] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.

[85] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. (2014).

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

[87] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515.

[88] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 931–948.

```
Processing the transactions of shard $S$ on replica $R$ :
Let txn_list be a transaction list storing the transactions from clients.
event Receive a transaction $T$ of shard $S^{'}$ from client do
    if $S \neq S^{''}$ then
        Redirect $T$ to the shard submitter of $S^{'}$.
        Return.
    Append $T$ to txn_list.

event Receive a batch $B$ of transaction from txn_list do
    outcomes $O$ = Execute ($B$)
    Deliver $B_r =< B, O, r >$ to DAG($B_r, r$)
event Receive a Shard reconfiguration $S^{'}$ do
    Start a new DAG
    Start to process the transactions of shard $S^{'}$.
```

Figure 20: Preplay.

```
event Receive a valid block $B_r$ of shard $S$ sent from replica $R$ at round $r$ from DAG do
    if $R$ is not the shard submitter of $S$ at round $r$ then
        Return invalid
    Build the dependency graph $G$ based on the read/write set $S$ in $B_r$.
    Execute the transactions simultaneously using $G$ and verify the results.
    if All the results are matched with the ones included in $B_r$ then
        Return valid
    else
        Return invalid

event Commit blocks $B$ from the committer at round $r$ from consensus do
    for Each block $b$ from round $r^{'}$ of sub-DAG $j$ in $B$ do
        if $b$ is a Shift block then
            num_committed_shift_block += 1
            Continue
        Update the values in the write sets to the storage.
    if num_committed_shift_block == $2f + 1$ then
        Reconfig the shard submitter to the next shard.
        num_committed_shift_block=0
```

Figure 21: Validate blocks.

```
Proposing a block $B$ at round $r$ on replica $R$ :

event Receive $2f + 1$ blocks $\{B\}$ at round $r - 1$ do
    need_shift = false
    if $\{B\}$ contains $f + 1$ Shift blocks then
        need_shift = true
    else
        for each replica $R$ do
            if Do not receive any block from $R$ after round $r - K$ then
                need_shift = true
            if $K^{'}$ blocks have been proposed then
                need_shift = true
    if need_shift = true and $R$ does not send $B_{shift}$ in the current DAG then
        Generate $B_{shift}$ and deliver to other replicas
    else
        Deliver block $B$ to other replicas
    Go to the next round
```

Figure 22: Broadcast the Shift block at round r if blocks from some replicas are missing from round $r - K$ or $K^{'}$ blocks have been proposed to make a periodical rotation.

Appendix A Concurrent Executor Implementation

We demonstrate the implementation of each component in *CE* in this section.

A.1 Execution on the Executors

```
1:  Let cc be the instance of $CC$.
2:  Let $result\_list$ be the execution outcomes for the transactions.
3:  Let $cid$ be the cid assigned by $CC$ if the transaction is read-only.
4:  Initial $cid = -1$.

5:  function Execute (Batch $b$)
6:      Initial the $result\_list$: $result\_list$.clear()
7:      Initial a new instance $CC$: $cc$.init()

8:      for Each transaction $T$ in Batch $b$ do
9:          Deliver $T$ to the executor to execute
10:     Wait for the results for all the transactions in
11:     Return $result\_list$

12: event Receive an abort of transaction $T$ from $CC$ do
13:     Abort $T$ if the executor of $T$ is alive
14:     Deliver $T$ to the executor to re-execute
15:     Commit $T$: cc.commit($T$)

16: event Receive a commit notification with <$T$, $cid$, $RES$, $V$> do
17:     result.append(<$T$, $cid$, $RES$, $V$>)

    Executor :
18: event Receive a transaction $T$ do
19:     Notify $CC$ to clean the state of $T$: cc.StartTxn($T$)
20:     Leverages eEVM to execute $T$

21: event Read key $K$, transaction $T$ do
22:     Return cc.read($K$, $T$)

23: event Write key $K$ with value $V$, transaction $T$ do
24:     Return cc.write($K$, $V$, $T$)
```

Figure 23: Executor Implementation.

Figure 23 shows the pseudocode code of the executors in *CE*. When *CE* receives a batch B of transactions, it initiates a set of executors and runs the eEVM [53], a tool to execute smart contracts, to execute the contract code. eEVM provides the read and write callback functions for the developers to implement their implementations to read and write the values for each key. When the read and write functions are triggered, *CE* will send the operation to *CC* (Line 22 and 24). If an abort of a transaction T is received from *CC* due to the conflict with other transactions, the execution will be aborted and T will be re-sent to the executor to re-execute. When eEVM completes the execution, the executor will commit the transaction (Line 15).

As discussed in section 7.2, the transaction is not actually committed after sending the commit request to *CC*. *CC* will send a notification after the transaction is finalized and *CE* can obtain the results (Line 17). Once all the transactions are committed, *CE* returns their scheduled orders (the commit order), read/write sets, and operation results.

A.2 Concurrency Controller

Concurrency Controller (*CC*) will receive four types of operations: Write, Read, Commit. *CC* will check if the transactions have been aborted before processing. The algorithm is shown in Figure 24 and Figure 25.

A.2.1 Write OP. When *CC* receives a write operation O_k from a transaction T with key K to update the value to V, If O_k is a new record, *CC* will add the necessary edges to the graph by linking all the non-write nodes (defined in section 8.1) to ensure the graph is valid as depicted in section 8.2. If adding the record failed due to a cycle detected, abort T by removing the node or processing cascading abort (Line 1 in Figure 25).

```
1:  Let $G$ be the dependency graph.
2:  Let $commit\_list$ be the list saving the transactions waiting for commit.

3:  function StartTxn (transaction $T$)
4:      Clean the abort state of $T$ if it is aborted.

5:  function Write (key $K$, value $V_{new}$, transaction $T$)
6:      if $T$ has been aborted then
7:          Return Fail
8:      Let $u$ be the node of transaction $T$.
9:      if If $u$ is a new node on $K$ then
10:         if $G$.AddNewWriteRecord($u$, $V_{new}$, $K$) returns Fail then
11:             AbortNode($u$)
12:             Return Fail
13:     Return Success

14: function Read (key $K$, transaction $T$)
15:     if $T$ has been aborted then
16:         Return Fail
17:     Let $u$ be the node of transaction $T$.
18:     if $u$ is a new node on $K$ then
19:         $u^{'}$ = $G$.AddNewReadRecord($u$, $K$)
20:     else
21:         $u^{'} = u$:
22:     if $u^{'}$ does not exist then
23:         AbortNode(u)
24:         Return Fail
25:     $V$ is the lastest updating value key $K$ on $u^{'}$
26:     Add a new record <$R$, $K$, $V$> to $u$
27:     Return $V$

28: function Commit (transactioin $T$)
29:     if $T$ has been aborted then
30:         Return Fail
31:     Let $u$ be the node of transaction $T$.
32:     if $u$ contains dependency edges in $G$ then
33:         $commit\_list$.append($T$)
34:     else
35:         CommitNode($u$)
36:         Remove $T$ from $commit\_list$
37:         for each transaction $T^{'}$ in $commit\_list$ do
38:             Let $u^{'}$ be the node of transaction $T$.
39:             if $u^{'}$ does not have dependency in $G$ then
40:                 Commit($T$)
```

Figure 24: *CC* implementation of processing transactions.

A.2.2 Read OP. Similar to the write OP, if the read operation O_k from T is a new record, *CC* returns the value and adds the necessary edges to the graph by linking all the write nodes. While adding the record, the graph will return the node $u^{'}$ the record refers to (Line 19 in Figure 24) which is the node T reads K from. If $u^{'}$ does not exist due to conflicts detected, abort T.

A.2.3 Commit. When receiving a commit request to inform the execution is done for transaction T, *CC* will place T to a pending list (*commit_list*) if there is any dependency for T in the graph (Line 33 in Figure 24). Otherwise, commit the nodes and generate the read/write sets RWS and the values of all the reads V. The commit order cid which is the scheduled order for the transactions is obtained. Then *CC* will notify the executors that transaction T has been committed, with the outcomes <T, cid, RWS, V>.

Upon a transaction T has been committed, it will check if there is any other transaction waiting in *commit_list* that it is allowed to commit as its dependency has been removed (Lines 37-40).

A.3 Dependency Graph

Figure 26 shows how the Dependency Graph is constructed in *CC*. When adding a write record to node u on key K, the graph will ensure all the non-write nodes on key K, which only reads the value on K, have the paths to u (Line 7). After adding the edges, a cycle check will be triggered to detect the conflict (Line 12).

```
function AbortNode (node $u$, transaction $T$)
    if $u$ is not a write node on every key then
        Remove($u$)
    else
        CasadingAbort($u$)
    Mark $T$ is aborted.
    Send notification to the executors.

function CommitNode (node $u$, transaction $T$)
    $RWS$ is the read/write sets
    $V$ is all the read results
    for Each record $< Type, Key, Value >$ in $u$ do
        $RES$.append($< Type, Key >$)
        if $Type = R$ then
            $V$.append($< Key, Value >$)
    $cid$ = Commit($T$)
    Notify $CE$ with <$T$, $cid$, $RES$, $V$>
```

Figure 25: *CC* implementation of committing and aborting transactions.

On the other hand, when adding a read record on node u, u tries to obtain the value from an incoming node x (Line 30). If no such node, a write node of key K will be searched or using the root node instead. When adding the edges from node x, some other edges will be added to ensure the graph is valid (Lines 23-25). Finally, returning the referring node x to *CC*.

Appendix B Thunderbolt Security Analysis

We provide the proofs for the security of Thunderbolt we have discussed in section 3.1. Suppose Thunderbolt generates a sequential order $SO = [T_1, \ldots, T_n]$ and produces an outcome $OUT = [OUT_1, \ldots, OUT_n]$ for a set of transactions. Let SE be the sequential execution in SO. Let the outcomes $OUT^{'}$ be the outcomes of SE.

B.1 Causal Ordering

Before giving proofs, we define notions for the causal ordering. Causal ordering of transactions in distributed systems is introduced by Lamport by defining a well-known "happened before" relation, denoted $\longrightarrow$ [51]. If A and B are two events, then $A \longrightarrow B$ if and only if one of the following conditions is true:

(1) A occurs before B in the same location;
(2) A is an outgoing message, and B corresponds to the response message;
(3) There is an existing event C that $A \longrightarrow C$ and $C \longrightarrow B$;

We extend the causal ordering from events to transactions in concurrency control to define a causal relationship between two transactions A and B:

Definition 6. If transaction A needs to be executed before B, $A \longrightarrow B$, if and only if B reads the data updated by A.

Definition 7. A and B can be executed concurrently only if $A \nrightarrow B$ and $B \nrightarrow A$.

Definition 8. If A has a causal relationship with B, either $A \longrightarrow B$ or $B \longrightarrow A$.

Definition 9. If $A \longrightarrow B$ and $B \longrightarrow C$, then $A \longrightarrow C$.

```
Let $root$ is the root node.

function AddNewWriteRecord (node $u$, value $V$, key $K$)
    $link\_to\_root = true$
    for each non-write node $v$ containing records on $K$ do
        if $v$ does not contain any outgoing edge on $K$ then
            /* No other nodes depending on $v$ on $K$ */
            Add edges($v$, $u$, $K$)
            $link\_to\_root = false$
    if $link\_to\_root == true$ then
        Add edges($root$,$u$,$K$) /* Link to root */
    else
        if Contain a cycle on $u$ then
            Return Fail
    Return Sucess

/* Find a write node $x$ on $K$ and depend on $x$ then return the value from $x$ */
function AddNewReadRecord (node $u$, key $K$)
    /* Find a incoming node $x$ on key $K$ */
    $x$ = GetIncomingNode($u$, $K$)
    if $x! = None$ then
        $x$ = GetWriteNodeToRead($u$, $K$)
    if $x! = None$ then
        Add edges($x$,$u$,$K$) /* Link to $x$ */
        for Each write node on $w$ key $K$ do
            if No path from $w$ to $x$ on $K$ then
                Add edges($w$,$x$,$K$)
        Return $x$
    Add edges($root$,$u$,$K$) /* Link to root */
    Return $root$

/* Find a incoming node $x$ on key $K$ */
function GetIncomingNode (node $u$, key $K$)
    if $u$ contains any incoming read node $v$ of $u$ on key $K$ then
        if CheckCycle($v$, $u$, $K$) == Fail then
            Return $v$
    if $u$ contains any incoming write node $v$ of $u$ on key $K$ then
        if CheckCycle($v$, $u$, $K$) == Fail then
            Return $v$
    Return None

/* Find a write node $x$ to read on key $K$ */
function GetWriteNodeToRead (node $u$, key $K$)
    for each write node $v$ containing records on $K$ do
        if CheckCycle($v$, $u$, $K$) == Fail then
            Return $v$
    Return None

/* Check if a read node $v$ can read values on node $u$ on key $K$ */
function CheckCycle (node $u$, node $v$, key $K$)
    if $u$ is not a write node on key $K$ then
        /* read values from a node without any update will not affect the graph */
        Return Fail
    for Each write node on $w$ key $K$ do
        /* need to ensure $u$ is the last update */
        if Contain a path from $v$ to $w$ then
            /* a cycle occurs $u \rightarrow v \rightarrow w \rightarrow u$ */
            Return Fail
    Return Success
```

Figure 26: Dpendency Graph Implementation.

B.2 Proof of Serializability

If Thunderbolt is serializable, $OUT = OUT^{'}$.

Definition 10 (Read-Complete). If T_i reads a value from T_j in the execution in Thunderbolt, T_i will also read the value from T_j in SE.

Definition 11 (Write-Complete). If T_i and T_j both write new values on K but T_i commits before T_j in the execution in Thunderbolt, T_i will also write the values on K before T_j when in SE.

Theorem 12. *Thunderbolt is both Read-Complete and Write-Complete if the dependency graph G is always valid.*

Proof. Firstly, if G is valid, we know that if there is a read node R_v^k reading a value from a write node W_u^k on key K, all the write nodes writing values on K either have a path to u or having a path from v to guarantee the correctness of read after write. Therefore, if transaction T_i reads values on T_j on key K, all other transactions writing values will not be assigned an order between T_i and T_j. Thus T_i will read the same value on T_j and Thunderbolt is Read-Complete.

Secondly, since SO is the commit order in Thunderbolt, if T_i commits before T_j, T_i will be assigned before T_j in SO. Thus, T_j will update the values after T_i in SE and Thunderbolt is Write-Complete. □

Theorem 13. *Thunderbolt is serializability iff Thunderbolt is both Read-Complete and Write-Complete.*

Proof. For any transaction T_i in SO, if T_i reads some values on the transactions $T_j \le T_i$ in Thunderbolt, T_i must read the same values in SE since Thunderbolt is Read-Complete. If T_i writes some new values, since Thunderbolt is Write-Complete, T_i will write the same values in SE and all the transactions $T_j < T_i$ have been committed. Thus, transactions will produce the same outcomes in Thunderbolt and SE: $OUT = OUT^{'}$. □

B.3 Proof of Safety

Thunderbolt produces the sequential order SO as well as the read-/write sets of each transactions. If the read/write set RS_i of transaction T_i overlaps the read/write set RS_j of transaction T_j and $T_i \rightarrow T_j$ in SO, T_j has a dependency on T_i. Therefore, if T_i dependents on T_j in one validator, other validators will have the same dependency. Finally, all the validators contain the same dependency graph generated by the read/write sets and SO. Thus, following the dependency graph to execute the transactions leading them to obtain the same outcomes.

B.4 Proof of Liveness

If all the replicas behave well, they will keep proposing the blocks in the same DAG. All the blocks proposed by each shard submitter will be committed eventually. When a malicious replica is detected, honest replicas will propose a Shift block. If less than $2f + 1$ Shift blocks are proposed, the DAG will not be switched and all the replicas will stay in the current DAG and keep proposing the new blocks. If there are $2f + 1$ Shift blocks, all the honest replicas will switch to the new DAG at the same round (section 2.2). After at least $2f + 1$ honest replicas have relocated to the new DAG, they are able to propose the new blocks.

If all the replicas behave properly, they will consistently propose blocks within the same DAG. Each shard submitter will eventually have the blocks they proposed committed Upon detection of a malicious replica, honest replicas will propose a Shift block. If fewer than $2f + 1$ Shift blocks are proposed, the DAG will remain unchanged, and all replicas will persist within the current DAG, continuing to propose new blocks. If there are $2f+1$ Shift blocks, all honest replicas will transition to the new DAG at the same round (section 2.2). Following the relocation of at least $2f + 1$ honest replicas to the new DAG, they will have the capability to propose new blocks.