MorphDAG: A Workload-Aware Elastic DAG-Based Blockchain

Shijie Zhang, *Student Member, IEEE*, Jiang Xiao, *Member, IEEE*, Enping Wu, *Student Member, IEEE*, Feng Cheng, *Student Member, IEEE*, Bo Li, *Fellow, IEEE*, Wei Wang, *Member, IEEE*, and Hai Jin, *Fellow, IEEE*

***Abstract*—*Directed Acyclic Graph* (DAG)-based blockchain represents a paradigm shift from conventional blockchains, which has the potential to drastically improve throughput performance through concurrent storage and executions. In practice, however, existing DAG-based blockchains fail to deliver such promises, often with limited throughput, high conflicts, and security vulnerabilities under dynamic workloads. The root causes are their unawareness of the workload characteristics of different workload sizes and skewed access patterns. In this article, we propose MorphDAG, the first workload-aware DAG-based blockchain that can significantly enhance throughput without compromising security and achieve elastic scaling under realistic workloads. We derive the theoretically optimal degree of storage concurrency to achieve high throughput while retaining system security as the workload size changes, while enabling fine-grained concurrency adjustment that accommodates a *Proof-of-Stake* (PoS)-based consensus protocol. We develop a dual-mode transaction processing mechanism that effectively resolves the conflicts brought by skewed access. We implement a prototype of MorphDAG and evaluate under real-world workloads. Extensive evaluations demonstrate that MorphDAG improves end-to-end throughput by up to 2.3× and 2.4× over state-of-the-art DAG-based blockchain systems AdaptChain and OHIE, respectively.**

***Index Terms*—DAG-based blockchains, workload characteristics, degree of storage concurrency, transaction processing.**

I. Introduction

BLOCKCHAIN technology has emerged as the backbone in a wide range of *decentralized applications* (dApps) [1], [2]. The diversified DApps put forward higher requirements for blockchain performance [3]. Recent studies from both industry [4], [5], [6] and academia [7], [8], [9], [10] present a paradigm shift from a single chain structure toward a *Directed Acyclic Graph* (DAG) structure for dramatic performance enhancement. Due to the inherent parallel structure, DAG-based blockchains show greater promise in supporting applications with high-performance demands [11].

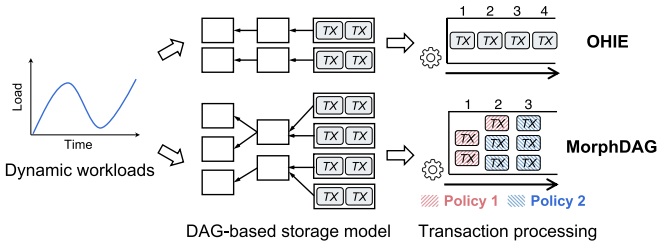


Fig. 1. Comparison of DAG-based blockchains (MorphDAG (our work) versus OHIE [7]).

While DAG-based blockchains have made progress on scaling consensus, they fail to incorporate the workload characteristics of real-world blockchain applications. We conduct a measurement study on the realistic workloads of Ethereum [12], the most prevailing blockchain system, in terms of workload size and data access. Our studies (Section II-C) present that Ethereum workloads exhibit two crucial characteristics: 1) unpredictable load changes that the load fluctuation does not follow a regular pattern, and 2) skewed access patterns that lead to data hotspots. This requires DAG-based blockchains to provide the *elasticity* guarantee that maintains high performance in spite of load variations and skewed data access.

Existing DAG-based blockchains [7], [8], [9], [13], however, are often configured *statically* with a workload-agnostic storage model and transaction processing policy, leading to performance degradation in the face of load changes and skewed data access. Specifically, prior systems adopt a *fixed degree of storage concurrency* to include transactions into a constant number of blocks that can be generated in parallel for on-chain consensus and storage. This leads to excessive latency as many transactions fail to be included in concurrent blocks when the load increases, and incurs unnecessary block propagation cost due to redundant concurrent blocks when the load reduces, as depicted in Fig. 1. On the other hand, a *single transaction processing policy* (e.g., serial processing or a single concurrency control mechanism) is typically employed to execute transactions for system state updates. A single policy cannot adapt to different conflict

Manuscript received 14 September 2023; revised 5 February 2024; accepted 19 March 2024. Date of publication 28 March 2024; date of current version 4 October 2024. This work was supported in part by National Key Research and Development Program of China under Grant 2021YFB2700700, and in part by Key Research and Development Program of Hubei Province under Grant 2021BEA164. Recommended for acceptance by Y. Tong. *(Corresponding author: Jiang Xiao.)*

Shijie Zhang, Jiang Xiao, Enping Wu, Feng Cheng, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: shijiezhang@hust.edu.cn; jiangxiao@hust.edu.cn; enpingwu@hust.edu.cn; fengcheng@hust.edu.cn; hjin@hust.edu.cn).

Bo Li and Wei Wang are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong (e-mail: bli@cse.ust.hk; weiwa@cse.ust.hk).

Digital Object Identifier 10.1109/TKDE.2024.3382743

situations caused by skewed data access. As a result, it either ignores the execution concurrency available on the cold data with few conflicts, or suffers from considerable transaction aborts due to severe conflicts on the hotspot data [13].

Supporting elasticity for a DAG-based blockchain under load variations and skewed data access faces a number of challenges. First, dynamically adjusting the degree of storage concurrency should achieve high throughput without compromising security under varying loads. Simply choosing a high degree of storage concurrency to include most transactions in the load would incur two issues: 1) the throughput improvement is limited as the existence of duplicate transactions included in concurrent blocks can neutralize the benefit brought by the high degree of storage concurrency; 2) it introduces security vulnerabilities since an adversary can obtain chances to generate malicious blocks on many parallel branches. Hence, selecting an appropriate degree of storage concurrency for the current load that enhances throughput while guaranteeing security is essential.

The second challenge is to address the performance bottleneck of transaction processing under skewed access patterns. An intuitive approach is to adopt different policies to process transactions accessing hotspot and cold data. In general, transactions accessing cold data can be processed with higher execution concurrency. However, skewed access patterns in DAG-based blockchains cause access explosion on hotspots due to the soaring number of transactions to be processed. This leads to considerable concurrent *read-modify-write* (RMW) operations [14] on hotspots, yielding more severe read-write conflicts than chain-based blockchains. Existing solutions either abort conflicting transactions [13], [15], [16] or conduct sequential execution [17], both of which will lead to significant performance degradation.

There have been several recent works aiming to improve the performance of DAG-based blockchains [7], [9], [10], [13]. OHIE [7] and Conflux [9] generate blocks in parallel with a fixed degree of storage concurrency, which cannot achieve elastic scaling under the dynamic workload size. Nezha [13] exploits concurrency in transaction processing, which reduces processing latency only in a synthetic workload (i.e., *Smallbank* benchmark). AdaptChain [10] adopts an exponential adjustment for the degree of storage concurrency and processes transactions serially, which suffers from security risks under a large degree of storage concurrency and limited throughput under skewed data access. To sum up, none of them address the elasticity of storage and transaction processing under realistic blockchain workloads.

In this paper, we focus on optimizing two key enablers of an elastic DAG-based blockchain: storage model and transaction processing policy. We present MorphDAG, the first workload-aware DAG-based blockchain that presents a comprehensive characterization of workloads in DAG-based blockchains and enables elastic storage and transaction processing. As depicted in Fig. 1, MorphDAG contains two innovations: an elastic DAG-based storage model and a dual-mode transaction processing mechanism. To support real-time workload awareness, MorphDAG implements a monitor module to perceive dynamic load changes and obtain the data access pattern through speculative execution.

To address the first challenge, we derive the elastic degree of storage concurrency theory that explores the impact of the degree of storage concurrency on system performance and security based on a few key insights. First, we observe that there exists a degree of storage concurrency that reaches the upper limit of throughput gains given the existence of duplicate transactions. Second, we observe that system security can be guaranteed with an extremely high probability as long as the number of branches remains within a safety threshold. To apply this theory to build up an elastic storage model under a mainstream permissionless consensus protocol *Proof-of-Stake* (PoS) [18], [19], we design a sortition-based concurrency adjustment mechanism that utilizes a *Verifiable Random Function* (VRF) to randomly select multiple block proposers based on the theoretical result that reaches the performance limit and runs the number of branches at the safety threshold to guarantee system security.

To address the second challenge, MorphDAG incorporates a novel dual-mode transaction processing mechanism that can effectively handle different conflict situations on the hotspot and cold data, respectively. To overcome the performance bottleneck of hotspots, MorphDAG exploits the key insight that committing incremental states of write operations can eliminate unnecessary reads during execution and devises an incremental state-based processing policy to reduce read-write conflicts. Aiming at the few conflicts on cold data, MorphDAG leverages a conflict detection-free processing policy to perform fast transaction ordering and execute transactions with higher execution concurrency.

We have implemented MorphDAG in Golang[1] and evaluated its performance in a geographically distributed environment using the realistic dataset from Ethereum. Our comparison between MorphDAG and state-of-the-art DAG-based blockchain systems (i.e., OHIE and AdaptChain) demonstrates that MorphDAG not only achieves up to 2.3× and 2.4× overall throughput improvement over AdaptChain and OHIE, respectively, but also achieves 4.3× speedup in transaction processing over them on average.

In a nutshell, the contributions of our work are summarized as follows.

- For the first time, we investigate the characteristics of real-world blockchain workloads and explore the crucial implications for DAG-based blockchains.
- We propose an elastic degree of concurrency theory and a sortition-based concurrency adjustment mechanism that achieves high throughput and security under load changes.
- We design a dual-mode transaction processing mechanism that efficiently processes transactions accessing hotspot and cold data.
- We implement a prototype of MorphDAG and evaluate it to demonstrate that MorphDAG outperforms state-of-the-art DAG-based blockchains in both throughput and latency.

The rest of this paper is organized as follows. Section II discusses the background and motivation behind MorphDAG. The system overview and workflow are presented in Section III.

[1] https://golang.org

Sections IV and V elaborate on the elastic storage model and the dual-mode transaction processing mechanism. Relevant security analysis is presented in Section VI. Section VII introduces the implementation of MorphDAG, and Section VIII evaluates its performance. Section IX discusses related work, and Section X concludes the paper.

II. Background and Motivation

In this section, we first describe the background of DAG-based blockchains and PoS-based consensus protocols. Then, we analyze the characteristics of the realistic blockchain workloads that motivate our main designs.

A. DAG-Based Blockchains

DAG-based blockchains have become a promising alternative to the conventional chain-based blockchains [12], [20] for improving scalability. The general workflow of a DAG-based blockchain consists of (1) *transaction inclusion* - transactions are included into multiple blocks by block proposers, (2) *transaction appending* - multiple blocks are then appended to the DAG in parallel, and (3) *transaction processing* - each node executes transactions of concurrent blocks to arrive at a consistent state. Throughout the workflow, the DAG-based storage model determines the number of transactions in the workload that can be effectively stored in the blockchain (i.e., transaction appending throughput), while transaction processing determines the latency of executing transactions for state persistence. Hence, the storage model and transaction processing are the two key factors impacting the performance of a DAG-based blockchain.

B. PoS-Based Consensus Protocols

Due to enhanced energy efficiency, *Proof-of-Stake* (PoS) has become the underlying consensus protocol of current mainstream public blockchains, such as Ethereum [12], which switched from *Proof-of-Work* (PoW) to PoS in 2022. In PoS, the block proposer election is primarily determined by the monetary value (i.e., stake) held by each node. However, unlike PoW that employs cryptographic puzzles to randomize election outcomes, PoS lacks this randomness, making it vulnerable to an adversary predicting the block proposer in advance. To prevent this, many PoS-based consensus protocols [12], [18], [19] adopt a *Verifiable Random Function* (VRF) [21] to insert the randomness in the election process.

VRF was first used in Algorand [18] to implement the sortition of block proposers, where each node can complete the sortition locally, and their sortition results can be verified by other nodes. A VRF mainly consists of two functions: `VRFRand` and `VRFVerify`, which are defined as follows:

- `VRFRand` takes as input a string u and a node's private key sk and outputs two values: a pseudo-random value $hash$ and a proof π. This function achieves the property of *pseudorandomness* [21] since $hash$ is indistinguishable from any random values to any node that does not know sk.

TABLE I
Ratio of Duplicate Transactions Under Varying Degree of Storage Concurrency

Degree of storage concurrency	10	20	30	40	50	60
Ratio of duplicate TXs (***random-based*** **[22]**)	36%	58%	69%	76%	81%	84%
Ratio of duplicate TXs (***fee-based*** **[12]**)	88%	94%	96%	97%	98%	98%

- `VRFVerify` takes as input four elements (pk, u, $hash$, π) and returns either 0 or 1, which enables any node that knows the public key pk to verify that the unique $hash$ corresponds to the input u, thereby ensuring the property of *unique provability* [21].

In Algorand, the system specifies a sortition target parameter τ to determine the expected number of selected block proposers in the current round. Each node utilizes `VRFRand` to generate a random value $hash$, and then checks if $hash/2^{len}$ (len denotes the bit-length of $hash$) falls within the probability interval D^l determined by the parameter τ. If it does, the sortition is considered successful. In this paper, we employ a VRF-based sortition process similar to Algorand, yet the sortition target τ is dynamically adjusted based on the degree of storage concurrency. The specific details are presented in Section IV-B.

C. Understanding the Characteristics of Realistic Workloads

Measurement Setup: We choose the largest and most representative account-based blockchain platform Ethereum [12], with massive transaction data.[2] We employ the dataset including transactions from block height 14,000,000 to 14,200,000 between January 2022 and February 2022, which is consistent with that used in Section VIII.

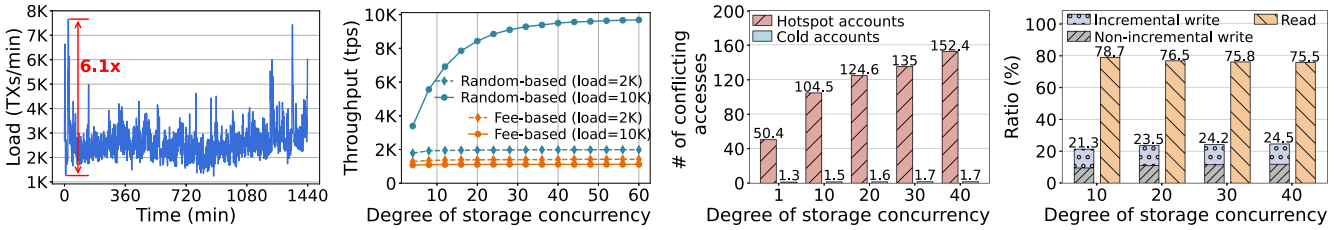
We start with measuring the impact of dynamic workload size on the DAG-based storage model. We monitor the workload size of Ethereum in one day. Fig. 2(a) shows that the workload size will surge (i.e., 7.6 K) and drop (i.e., 1.2 K) multiple times in an irregular pattern in a day.

Observation 1: The workload size exhibits unpredictable changes, and the peak size is about $6.1\times$ the trough.

Implication 1: The degree of storage concurrency should be elastically scaled to the dynamic workload size for throughput improvement.

The inherent parallelism of the DAG-based blockchain leads to substantially duplicate transactions since different block proposers cannot perceive which transactions have been included. We employ two common transaction inclusion strategies (i.e., fee-based [12] and random-based [22]) and investigate the ratio of duplicate transactions with varying storage concurrency under a workload size of 10 K. Table I shows that the ratio of duplicate transactions rises with the degree of storage concurrency increases under both strategies, where the fee-based strategy (i.e., prioritize transactions with higher transaction fees) yields

[2]Existing DAG-based blockchains still employ Ethereum workloads for test because of lacking publicity available transaction data.



(a) Dynamic workload size (b) Appending throughput under varying storage concurrency (c) Skewed access pattern (d) Proportion of read and write operations during accesses

Fig. 2. The characteristics of real-world blockchain workloads.

a higher ratio than the random-based strategy since the transactions with the top fees in each node's transaction pool are roughly the same. We further study the impact of duplicate transactions on appending throughput under two workload sizes. As shown in Fig. 2(b), under a small workload size (i.e., 2 K), both strategies can achieve an optimal throughput that adapts to the current load with a small degree of storage concurrency. However, the increase in the degree of storage concurrency would cause a waste of resources (e.g., network and computing). Under a large workload size (i.e., 10 K), the throughput gain brought by the increase in the degree of storage concurrency is bottlenecked in both strategies.

Observation 2: The existence of duplicate transactions during the parallel transaction inclusion of DAG-based blockchains will bottleneck the performance gain brought by the increased degree of storage concurrency.

Implication 2: DAG-based blockchains gain limited throughput benefits from simply enlarging the degree of storage concurrency, which may incur security risks since an adversary gains more chances to generate malicious blocks on multiple branches. Hence, it is essential to find an appropriate degree of storage concurrency that achieves high throughput without compromising security.

Next, we investigate the data access characteristic of workloads to study its impact on transaction processing. Specifically, we replay 40,000 Ethereum transactions in a DAG-based blockchain, including transferring tokens and invoking smart contracts, and record the average number of conflicting accesses on the hotspot and cold accounts. We regard the top $k\%$ most frequently accessed accounts as hotspots [23] (k is set to 1 here according to the calibration in Section VIII-D). Fig. 2(c) reports that, compared with a chain-based blockchain whose degree of storage concurrency equals 1, the conflicting accesses of hotspot accounts are much higher than that of cold accounts in a DAG-based blockchain, and the data contention becomes more severe as the degree of storage concurrency increases.

Observation 3: DAG-based blockchain falls short of fully leveraging the degree of storage concurrency since the substantial concurrent accesses on hotspots lead to severe data contention, which significantly degrades transaction processing performance.

Implication 3: The transaction processing policy of DAG-based blockchains should be tailored for diversified data contention situations of hotspot and cold accounts.

We further study the read and write operations among concurrent accesses to identify the root cause of conflicts. Fig. 2(d) demonstrates that the proportion of read and write operations of Ethereum workloads is approximately 3:1. Except for read-only operations, substantial read operations arise from the *read-modify-write* (RMW) pattern [14] of write operations in Ethereum, which requires fetching the account state before writing. Thus, write operations are frequently accompanied by considerable concurrent reads, causing severe read-write conflicts.

Furthermore, an interesting observation in Fig. 2(d) is that over 50% of RMW operations are *incremental writes*. For instance, in a simple funds transfer, the increase in the payee's account balance is an *incremental write* operation. The *incremental write* operation can be executed without the need to read the account state if we commit the increment of state rather than updating the state.

Observation 4: Severe data contention on hotspot accounts stems from read-write conflicts caused by substantial concurrent read and write operations.

Implication 4: The key to mitigating read-write conflicts on hotspots is to reduce the proportion of reads during performing RMW operations. We can transform a part of conflicting RMW operations into committing incremental states to reduce the number of reads and improve execution concurrency.

III. Overview of MorphDAG

A. Goals and Assumptions

MorphDAG is a permissionless DAG-based blockchain that operates consecutive block generation epochs. To enable parallel block generation, MorphDAG employs a PoS-based consensus that selects a number of block proposers among nodes proportionally to their stake in each epoch. MorphDAG adjusts the degree of storage concurrency by changing the number of block proposers. To determine the canonical blockchain history, MorphDAG adopts the GHOST protocol that is commonly adopted in DAG-based blockchains [9], [10], [24] to select the heaviest one (i.e., with the largest number of blocks) when encountering divergent histories. To support Turing-complete smart contract execution, MorphDAG employs an account-based model [12] for storing system states.

MorphDAG achieves the following two goals: (1) *Elastic storage:* MorphDAG can elastically adjust the degree of storage

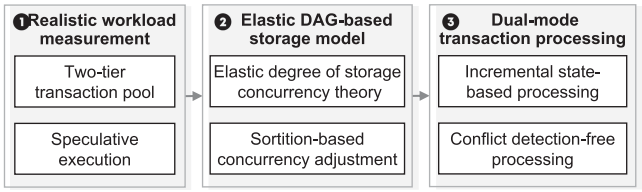


Fig. 3. MorphDAG architecture.

concurrency according to the dynamic workload size and reach an optimal level of throughput under the current load while retaining system security. (2) *Elastic transaction processing:* MorphDAG can process transactions in different policies to adapt to different data contention caused by skewed data access, and efficiently execute transactions accessing hotspots without yielding any transaction aborts.

MorphDAG adopts the same threat model and network assumptions as prior works [7], [8], [9]. Suppose the adversary cannot corrupt a majority of parties and holds a σ_a fraction of stake units in the system, where σ_a is below $1/2$. The adversary attempts to forge the heaviest branch after a given block to become the canonical blockchain history, i.e., history-rewrite attacks [19]. However, the adversary cannot destroy standard cryptographic primitives, e.g., public-key signatures and hash functions. Honest nodes strictly follow the protocol, and all messages among honest nodes can arrive within a maximum propagation latency of δ.

B. System Workflow

The workflow of MorphDAG includes the necessary three steps mentioned in Section II-A. Besides, MorphDAG consists of the workload measurement process before transaction inclusion to provide real-time workload awareness. Fig. 3 shows the overall architecture of a MorphDAG node. We now describe MorphDAG's workflow and core components: ❶ *Realistic Workload Measurement:* Each node locally measures the workload from their transaction pools before block generation. The measurement module comprises a two-tier transaction pool and a speculative execution mechanism.

Two-Tier Transaction Pool: MorphDAG provides a two-tier transaction pool to perceive variations in workload size. Specifically, the *pending pool* is used to store all newly input transactions, while the *queue pool* is used to store transactions to be included into blocks. The current workload size can be obtained by observing the scale of the *pending pool*.

Speculative Execution: MorphDAG fetches all accessed accounts and access patterns of each transaction through speculative execution based on the latest system states, and maintains the statistics of account access frequency to select top $k\%$ most frequently accessed accounts as hotspots. The speculative execution of transactions can be concurrent since it only fetches account states rather than modifying them.

❷ *Elastic DAG-Based Storage Model (Section IV):* The construction of the elastic DAG-based storage model consists of transaction inclusion and appending. Each node locally derives a theoretical degree of storage concurrency based on the obtained workload size, and adjusts the sortition target to perform the sortition of block proposers and the parallel block generation (if selected). The transaction workload is then appended to the DAG in the form of concurrent blocks.

Elastic Degree of Storage Concurrency Theory (Section IV-A): The theory of the elastic degree of storage concurrency is derived from the perspectives of performance and security. Specifically, we analyze the transaction inclusion behavior of nodes to obtain the performance threshold that causes a throughput gain bottleneck. Besides, we derive the safety threshold by analyzing the impact of the degree of storage concurrency on system security.

Sortition-Based Concurrency Adjustment (Section IV-B): During the sortition of block proposers, each node computes a VRF and produces a pseudo-random output to indicate whether it is chosen as a block proposer. According to the performance threshold of the degree of storage concurrency, each node adjusts the sortition target in the VRF to produce the expected number of block proposers for parallel block generation. The system runs with a constant number (i.e., safety threshold) of branches composed of concurrent blocks to guarantee high performance and security.

❸ *Dual-Mode Transaction Processing (Section V):* After a number of concurrent blocks are appended to the DAG, each node fetches transactions in all concurrent blocks and removes duplicates. These transactions are then efficiently processed in dual modes under different data contention situations to achieve system state transitions.

Incremental State-Based Processing (Section V-A): Aiming at the severe data contention on hotspot accounts, MorphDAG performs prioritized transaction ordering and conflict detection based on access patterns before transaction execution, and commits incremental states to reduce considerable read-write conflicts on hotspot accounts.

Conflict Detection-Free Processing (Section V-B): Due to few conflicting accesses towards cold accounts, MorphDAG removes tedious conflict detection and achieves fast transaction ordering and highly concurrent transaction execution.

IV. Elastic DAG-Based Storage Model

In this section, we detail our elastic degree of storage concurrency theory and how MorphDAG builds an elastic storage model based on the theory under the dynamic workload size. The overall process is depicted in Fig. 4. Important notations are listed in Table II.

A. Elastic Degree of Storage Concurrency Theory

Performance Threshold: We first explore the degree of storage concurrency that enables high appending throughput. We investigate the relationship between the degree of storage concurrency and throughput by analyzing the transaction inclusion behavior of each node. According to Fig. 2(b), we choose the random-based transaction inclusion strategy [22] due to its lower ratio of duplicate transactions and higher appending throughput.

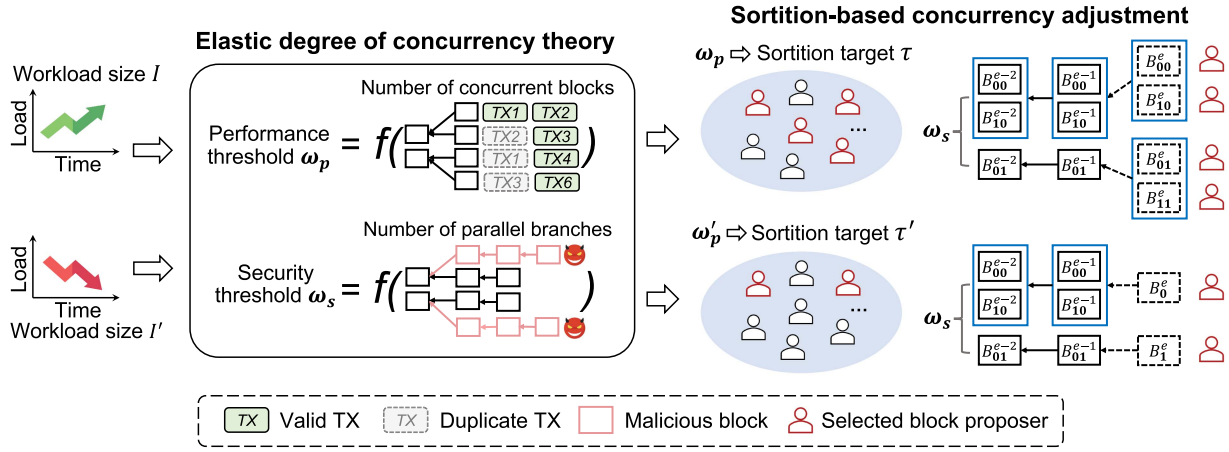


Fig. 4. Illustration of how MorphDAG's storage model elastically scales to the dynamic workload size.

TABLE II
SUMMARY OF IMPORTANT NOTATIONS

Notation	Description	Notation	Description
δ	Propagation latency	p_i	Probability that T_i is included into blocks
Δ	Epoch time	ζ	Throughput gain
$\mathbf{W}_\Delta$	Workload in Δ	z	# of successor blocks for confirmation
I_Δ	Workload size in Δ	ϵ	Security parameter
μ	Workload input rate	σ_j	Stake fraction of node j
λ	Propagation rate	M	# of stake units in the system
$\mathbf{T}_j$	TX pool of node j	sk_j	Private key of node j
n	Block size	pk_j	Public key of node j
s_j	TX pool size of node j	τ	Sortition target
ω	Degree of storage concurrency	N	# of nodes in the system

We assume that the time Δ of a block generation epoch is at least c (c is a constant) times δ for high system security [7]. The workload $\mathbf{W}_\Delta$ contains I_Δ transactions, i.e., $\mathbf{W}_\Delta = \{TX_i | 1 \leq i \leq I_\Delta\}$. Assuming that the workload input process follows the Poisson distribution [25] with the rate $\mu = I_\Delta/\Delta$. We investigate the case where transactions in $\mathbf{W}_\Delta$ are included into concurrent blocks during Δ.

To simplify the analysis, we assume that each transaction pool $\mathbf{T}_j$ only contains transactions from $\mathbf{W}_\Delta$. Due to network propagation delays, some transactions from $\mathbf{W}_\Delta$ may not be included in $\mathbf{T}_j$ during Δ. Hence, whether a transaction TX_i can be included into concurrent blocks depends on two factors: one is that TX_i is included in a transaction pool $\mathbf{T}_j$, and the other is that TX_i is selected from $\mathbf{T}_j$ by the block proposer j. We denote the probability of occurrence of these two factors by $p_i^{\mathbf{T}}$ and $p_i^{\mathbf{B}}$, respectively.

The probability $p_i^{\mathbf{T}}$ is equivalent to the probability that TX_i is received by a node within Δ, which is relevant to the time when TX_i is initiated. Since the network propagation has an upper limit of latency δ, as long as the transaction initiation time t is within $\Delta - \delta$, TX_i can be received by all nodes and included in their transaction pools at the current epoch. Otherwise, TX_i may not reach all nodes in Δ with a certain probability $1 - \gamma(t)$ if its initiation time t exceeds $\Delta - \delta$. By combining the above cases, we can deduce the probability density function $p_i^{\mathbf{T}}(t)$:

$$p_i^{\mathbf{T}}(t) = \begin{cases} 1, & t \leq \Delta - \delta \\ \gamma(t), & \Delta - \delta < t < \Delta \\ 0, & t \geq \Delta \end{cases} \tag{1}$$

Let us consider the case that $t \in (\Delta - \delta, \Delta)$. According to the distribution of message propagation delay in the P2P network, the time interval for nodes to receive messages conforms to exponential distribution [26]. Therefore, the probability that TX_i whose initiation time t is within $(\Delta - \delta, \Delta)$ can be received by all nodes is $\gamma(t) = 1 - e^{-\lambda(\Delta - t)}$.

Since the workload input process follows the Poisson process with the rate $\mu = I_\Delta/\Delta$, the time when the transaction TX_i is sent has a Gamma distribution. Based on (1) and the expression of $\gamma(t)$, we can utilize the probability of the transaction initiation time distribution to obtain the expression of $p_i^{\mathbf{T}}$, which follows that:

$$p_i^{\mathbf{T}} = \int_0^{\Delta-\delta} \frac{\mu^i t^{i-1} e^{-\mu t}}{\Gamma(i)} dt + \int_{\Delta-\delta}^{\Delta} \frac{\mu^i t^{i-1} e^{-\mu t}}{\Gamma(i)} \cdot (1 - e^{-\lambda(\Delta - t)}) dt \tag{2}$$

On the other hand, $p_i^{\mathbf{B}}$ is relevant to the random-based transaction inclusion strategy. Suppose that all concurrent blocks maintain the same block size n, we can obtain the expression of $p_i^{\mathbf{B}}$, i.e., $p_i^{\mathbf{B}} = n/s_j$. Since s_j cannot be calculated precisely, here we can utilize the expected value of transaction pool size to obtain the approximation of s_j, i.e., $\mathbb{E}(s_j) = \sum_{i=1}^{I_\Delta} p_i^{\mathbf{T}}$. Furthermore, the probability that transaction TX_i is included into at least one block in the current epoch is relevant to $p_i^{\mathbf{T}}$ and $p_i^{\mathbf{B}}$. It follows that:

$$p_i = 1 - (1 - p_i^{\mathbf{T}} \cdot p_i^{\mathbf{B}})^{\omega} \tag{3}$$

According to the expression of p_i (3), we can calculate the expected value of effective throughput $\mathbf{TPS}_\Delta$ within Δ based

on p_i. It follows that:

$$\mathbb{E}(\mathbf{TPS}_\Delta) = \sum_{i=1}^{I_\Delta} \frac{1-(1-p_i^{\mathbf{T}} \cdot p_i^{\mathbf{B}})^\omega}{\Delta} \tag{4}$$

To understand the throughput gain bottleneck incurred by duplicate transactions, we define ζ as the throughput gain brought by the increase in ω. ζ is the partial derivative of $\mathbb{E}(\mathbf{TPS}_\Delta)$, i.e., $\zeta = \partial\mathbb{E}(\mathbf{TPS}_\Delta)/\partial\omega$. The definition of throughput gain bottleneck is as follows.

Definition 1. (Throughput Gain Bottleneck): The throughput gain is bottlenecked if the current throughput gain ζ is less than or equal to a constant value ζ_0, i.e., $\zeta \leq \zeta_0$.

The smallest degree of storage concurrency ω_p leading to the bottleneck (i.e., performance threshold) can enable high throughput with the lowest propagation cost of concurrent blocks. Under the throughput gain bottleneck ($\zeta \leq \zeta_0$), we can obtain ω_p based on the expression of ζ:

$$\omega_p = \arg\max_{\omega} \frac{-\sum_{i=1}^{I_\Delta} \ln \upsilon_i \cdot \upsilon_i^\omega}{\Delta} \tag{5}$$

where $\upsilon_i = 1 - np_i^{\mathbf{T}} / \sum_{i=1}^{I_\Delta} p_i^{\mathbf{T}}$.

Safety Threshold: Typically, the current degree of storage concurrency determines the number of branches. Although ω_p can achieve high throughput, a large degree of storage concurrency would result in more chances for the adversary to forge malicious blocks on multiple branches. To make it difficult for the adversary to generate consecutive malicious blocks on a certain branch, we stipulate that newly created blocks will be randomly appended to one of the existing blocks, similar to [7], [10]. However, if one of the branches is controlled by the adversary, the valid transaction history of this branch will be rewritten. Therefore, we need to find a safety threshold for the degree of storage concurrency that does not compromise system security.

Let us consider the case that the adversary attempts to launch a history-rewrite attack on a specific branch. The race between the adversary and honest nodes to create blocks is actually a race of their stake fraction. Hence, on any branch, the ratio of the block creation rate of the adversary to that of honest nodes in an epoch is $\frac{\sigma_a}{1-\sigma_a}$. We assume the probability that the adversary successfully creates the heaviest malicious branch to subvert the valid history on a given branch within z confirmation block intervals is $\rho^{single}(z)$. We follow the analysis presented in [27], which indicates that the adversary's process of generating blocks has a negative binomial distribution. It follows that:

$$\rho^{single}(z) = 1 \sum_{m=0}^{z-1} ((1-\sigma_a)^m \sigma_a^z - \sigma_a^m (1-\sigma_a)^z) \binom{m+z-1}{m} \tag{6}$$

Furthermore, we can deduce the probability $\rho^{DAG}(\omega)$ indicating that there exist branches controlled by the adversary, which follows that:

$$\rho^{DAG}(\omega) = 1 - (1-\rho^{single}(z))^\omega \tag{7}$$

Suppose the system security parameter is ϵ, i.e., the probability of valid branch histories being rewritten that the system can tolerate. When the probability $\rho^{DAG}(\omega)$ is less than or equal to ϵ, the security of the system can be guaranteed. Thus let $\rho^{DAG}(\omega) = \epsilon$, we can obtain the safety threshold of the degree of storage concurrency ω_s:

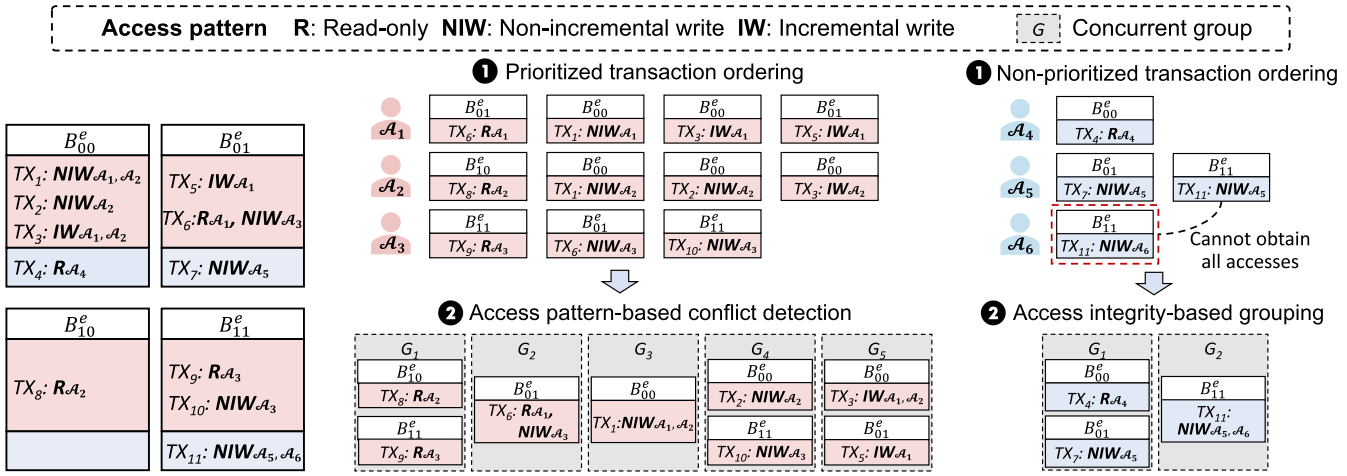
$$\omega_s = \log_{1-\rho^{single}(z)}(1-\epsilon) \tag{8}$$

If ω_s is used as the maximum degree of storage concurrency of the system for high security, system performance would be limited at a large load since the benefits of ω_p cannot be exploited (e.g., ω_p is greater than ω_s at this time). Below, we elaborate on how to apply ω_p and ω_s simultaneously for concurrency adjustment to achieve high throughput while guaranteeing security.

B. Sortition-Based Concurrency Adjustment

Sortition Target Adjustment: As depicted in Fig. 4, unlike the conventional VRF-based sortition [18] that adopts a fixed value of τ, each node in MorphDAG will first adjust the target τ before sortition to make the actual number of selected block proposers close to the theoretical degree of storage concurrency ω_p. Specifically, the sortition process regards each stake unit of a node as a sub-user and selects each sub-user with a probability $p_s = \frac{\tau}{M}$. Hence, the probability interval D^l determines the probability that the node has l selected sub-users. However, in this case, a node with a high stake fraction may have a chance to generate multiple blocks in an epoch, incurring the risk of *noting at stake* attacks [18], [19]. To prevent such security risks, we stipulate that each node will be selected as the block proposer only once in each epoch, no matter how many sub-users are chosen. Thus, D^l should indicate the probability that at least one sub-user is chosen, i.e., $D^l = 1-(1-p_s)^{\sigma_j M}$. Accordingly, the expected number of selected block proposers in an epoch is $\sum_{j=1}^{N} 1-(1-p_s)^{\sigma_j M}$. We can further simplify this expression by the limit: $N - \sum_{j=1}^{N} e^{-\tau\cdot\sigma_j}$. To render the number of generated concurrent blocks close to ω_p, each node j adjusts τ such that the equation $\omega_p = N - \sum_{j=1}^{N} e^{-\tau\cdot\sigma_j}$ holds. Then, each node performs $\texttt{VRFRand}(sk_j, seed)$ to derive an output $< hash, \pi >$ containing a pseudo-random value $hash$ and a proof π, where the input parameter $seed$ is publicly known and updated for each epoch by computing $\texttt{VRFRand}$ with the $seed$ of the previous epoch. Nodes with $hash/2^{len}$ falling within D^l will be selected as block proposers.

Block Generation and Verification: Each selected block proposer takes the random-based transaction inclusion strategy to pick transactions from their transaction pools with a probability $p_i^{\mathbf{B}}$. Then, each block proposer generates a block filled with picked n transactions. The necessary verification information is stored in the block header. In addition to the basic information (e.g., timestamp, transaction root, state root, and previous block hash), the block root and the VRF verification data are also filled into the block header. Specifically, since a block proposer does not know which branch a new block will extend until it finishes block creation, it computes a Merkle tree root as the input of block creation by using hash values of all the blocks in the last epoch. For the VRF verification data, a block proposer includes



(a) Access patterns on hotspot and cold accounts

(b) Incremental state-based processing policy

(c) Conflict detection-free processing policy

Fig. 5. Illustration of how MorphDAG performs dual-mode transaction processing.

the output $< hash, \pi >$ into the block header. To prevent the adversary from falsifying the target τ to become a block proposer, we stipulate that the newly observed workload size I_Δ is hard-coded into the block header. During block verification, each node derives the correct target τ according to I_Δ stored in the block header and computes $\texttt{VRFVerify}(pk_j, seed, hash, \pi)$ to verify $hash$ and check if $hash/2^{len}$ falls within D^l.

Aggregation-Based Block Attachment: To guarantee high performance and security, we devise an aggregation-based block attachment mechanism so that the system can generate ω_p concurrent blocks while maintaining the number of parallel branches at ω_s. Specifically, the system starts with ω_s concurrent blocks. When ω_p is greater than ω_s, multiple blocks are aggregated to connect predecessor blocks and can similarly be collectively connected by successor blocks. As depicted in Fig. 4, each new block in epoch e tries to find its predecessor with the same last $\lceil \log_2 \omega_s \rceil$ bits of the hash value and forms a connection. Note that when multiple previous blocks form an aggregation, each new block will identify their common tail. If not found, we keep lowering one bit of the block hash until each block finds the first predecessor with the same tail. If multiple blocks find the same predecessor, they will be aggregated to connect the predecessor. For instance, $(\mathbf{B}^e_{00}, \mathbf{B}^e_{10})$ connects $(\mathbf{B}^{e-1}_{00}, \mathbf{B}^{e-1}_{10})$ with the same tail 0. Therefore, the system always runs with ω_s parallel branches, and each branch follows the GHOST protocol, i.e., each block in the branch counts as a confirmation of the predecessor block.

Sortition Fairness: Since each node independently observes the workload size and locally calculates the sortition target τ, the probability of becoming a block proposer D^l can vary among them. However, we argue that the difference in D^l among nodes with identical stake fractions is negligible. First, the adversary cannot falsify τ to inflate D^l since honest nodes can detect malicious blocks by computing the correct D^l via I_Δ hard-coded in the block header. Besides, the range of each node's transaction pool size s_j used for calculating ω_p is $[I_{\Delta-\delta}, I_\Delta]$, and the difference between s_j is capped at I_δ. For instance, if two nodes with an equal stake fraction of 5% have a difference of 1000 in the observed workload size, the resultant difference in ω_p is 5, and in D^l, merely 2%.

V. Dual-Mode Transaction Processing

In this section, we introduce a dual-mode mechanism for efficient transaction processing. Fig. 5 depicts how the dual-mode mechanism works. The dual-mode mechanism takes as input the unique transactions in all concurrent blocks in the current epoch e, and relies on our workload measurement module to classify all transactions into those accessing hotspot accounts (TX^H) and those only accessing cold accounts (TX^C). Subsequently, we adopt two different transaction processing policies to process TX^H and TX^C.

A. Incremental State-Based Processing

Incremental State: According to *Implication 4* (Section II-C), over half of RMW operations are *incremental writes*. We can take the incremental state as the execution result to avoid unnecessary reads during the execution of *incremental writes*. It offers promising opportunities for reducing read-write conflicts between TX^H and enabling concurrent execution of write transactions. Committing incremental state values can reduce the proportion of reads, however, it enlarges the read overhead. Specifically, when a read operation intends to fetch the latest state of an account, it requires adding up all incremental values. To ensure read operations can be executed before *incremental write* operations, we employ one single thread as the ordering manager to perform transaction ordering before execution. Algorithm 1 describes pseudo-codes of the processing policy for TX^H.

Prioritized Transaction Ordering: We employ a transaction set $\texttt{TXSet}^H_r$ for each hotspot account $\mathcal{A}^H_r$ to store all transactions

Algorithm 1: Incremental State-Based Processing.

Input: Transactions accessing hotspot accounts TX^H

1: ▷ **Ordering manager:**
2: **for** each hotspot account $\mathcal{A}_r^H$ **do**
3: $\texttt{TXSet}_r^H \leftarrow IntializeTXSet(TX^H, \mathcal{A}_r^H)$
4: $Ordering(\texttt{TXSet}_r^H)$
5: **end for**
6: $G^H \leftarrow FormConcurrentGroup(\texttt{TXSet}^H)$
7: **for** each group G_m^H in G^H **do** ▷ except **IW** transactions
8: **for** each TX_i^H in G_m^H **do**
9: $Result \leftarrow CheckMapping(\texttt{M}_{\texttt{Acc}\rightarrow\texttt{R}}, \texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}, \texttt{M}_{\texttt{Acc}\rightarrow\texttt{IW}})$
10: **if** $Result == True$ **then**
11: $InsertMapping(TX_i^H)$
12: $G_m^H.delete(TX_i^H)$
13: **end if**
14: **end for**
15: $G^H.reorganizeGroup(\texttt{M}_{\texttt{Acc}\rightarrow\texttt{R}}, \texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}, \texttt{M}_{\texttt{Acc}\rightarrow\texttt{IW}})$
16: $Clear(\texttt{M}_{\texttt{Acc}\rightarrow\texttt{R}}, \texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}, \texttt{M}_{\texttt{Acc}\rightarrow\texttt{IW}})$
17: **if** G_m^H contains any TX_i^H **then**
18: **goto line 8**
19: **end if**
20: **end for**
21: ▷ **Thread scheduler:**
22: **for** each group $G_{m'}^H$ in G^H **do**
23: **for** each TX_i^H in $G_{m'}^H$ **do**
24: $T_i \leftarrow AssignIdleThread()$
25: $Updates \leftarrow T_i.execute(TX_i^H)$ ▷ run in parallel
26: $T_i.commitToStateDB(Updates)$ ▷ run in parallel
27: **end for**
28: **end for**

accessing it for ordering (lines 2-5). Transaction access patterns are categorized by *read-only* (**R**), *non-incremental write* (**NIW**), and *incremental write* (**IW**). **NIW** performs RMW operations, e.g., balance deduction operations require checking the balance at first. In each $\texttt{TXSet}_r^H$, transactions are ordered by **R**, **NIW**, and **IW** to avoid read operations on incremental states. The order between transactions of the same access pattern abides by block hash values and transaction IDs. Specifically, transactions with smaller block hashes and transaction IDs have higher ranks. For instance, as shown in Fig. 5(a) and (b), TX_3 is ranked after TX_6 and TX_1 according to the access patterns, and ahead of TX_5 due to its smaller block hash and transaction ID.

Access Pattern-Based Conflict Detection: To support concurrent execution, we form non-conflicting transactions into a concurrent group G_m^H (line 6). Transactions within a G_m^H are executed concurrently, while transactions between groups are executed serially. In particular, for **R** and **IW** transactions, those with the same access pattern can be assigned to the same G_m^H. On the contrary, for **NIW** transactions, only those accessing different accounts can be assigned to the same G_m^H.

TABLE III
MAPPING RULES FOR FINE-GRAINED CONFLICT DETECTION

Access pattern	Mapping rules
R	$\texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}[\mathcal{A}_r] = \emptyset \wedge \texttt{M}_{\texttt{Acc}\rightarrow\texttt{IW}}[\mathcal{A}_r] = \emptyset$
NIW	$\texttt{M}_{\texttt{Acc}\rightarrow\texttt{R}}[\mathcal{A}_r] = \emptyset \wedge \texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}[\mathcal{A}_r] = \emptyset \wedge \texttt{M}_{\texttt{Acc}\rightarrow\texttt{IW}}[\mathcal{A}_r] = \emptyset$
IW	$\texttt{M}_{\texttt{Acc}\rightarrow\texttt{R}}[\mathcal{A}_r] = \emptyset \wedge \texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}[\mathcal{A}_r] = \emptyset$

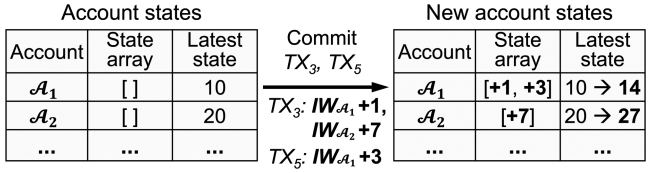


Fig. 6. An example of **IW** transaction committing.

However, transactions in the same group may access multiple accounts in different patterns and cause potential conflicts. Thus, we design a fine-grained conflict detection approach based on account-transaction mapping rules. Specifically, we devise three mapping structures based on the access patterns: $\texttt{M}_{\texttt{Acc}\rightarrow\texttt{R}}$, $\texttt{M}_{\texttt{Acc}\rightarrow\texttt{NIW}}$, and $\texttt{M}_{\texttt{Acc}\rightarrow\texttt{IW}}$. Within a G_m^H, we check each accessed account $\mathcal{A}_r$ (including cold accounts as some TX^H may access them) and the access patterns of each transaction and then map the transaction if it abides by all the mapping rules shown in Table III (lines 8-14). When a round of mapping is completed, the transactions in the mapping structures can be reassigned to a new concurrent group without conflicts (lines 15-19). It does not require conflict detection in the group of **IW** since transactions containing other access patterns have already been assigned to the previous groups. As depicted in Fig. 5(b), transactions accessing hotspots are assigned to five concurrent groups, where TX_6 is assigned to G_2 due to a read-write conflict between its **NIW** and the **R** of TX_9.

Transaction Execution and Committing: In each new group $G_{m'}^H$, we set up an independent thread per transaction for concurrent execution. Due to the varying number of transactions in different groups, we establish a thread scheduler to dynamically assign execution tasks to idle threads (line 24). After the execution of a transaction, the thread commits state updates of this transaction to `StateDB` (lines 25-26), an in-memory database storing account states, which is similar to Ethereum [12]. After the execution of transactions from all concurrent blocks is completed, `StateDB` will write all state updates to the on-disk database for state persistence.

To support the committing of **IW** transactions, we devise an `array` for each account state to cache all committed incremental states in `StateDB`. Fig. 6 shows that the incremental states of TX_3 (i.e., "+1", "+7") and TX_5 (i.e., "+3") are cached into the corresponding `array` in `StateDB`. `StateDB` adds up all incremental states in each `array` to update the latest state in two cases: i) when `StateDB` receives a read request from a transaction; ii) after all state updates in an epoch are committed to `StateDB`.

For **NIW** transactions, their execution results directly update the latest state in `StateDB` to ensure that subsequent transactions can retrieve the latest account state.

B. Conflict Detection-Free Processing

Unlike hotspots, data contention is far less severe on cold accounts due to few data accesses (*Observation 3* (Section II-C)). It does not require tedious ordering and conflict detection, instead, the principle is to ensure high execution concurrency with low ordering overhead. Similar to the processing policy towards TX^H, we set up a single-threaded ordering manager and a thread scheduler for processing TX^C. The operations of the thread scheduler and the concurrent threads are the same as those shown in Algorithm 1. However, the ordering manager does not require performing complex conflict detection.

Access Integrity-Based Grouping: The ordering manager maintains a transaction set $\texttt{TXSet}_r^C$ for each cold account $\mathcal{A}_r^C$ and directly sorts transactions in each $\texttt{TXSet}_r^C$ by their block hashes and transaction IDs without considering access patterns. It then assigns TX^C to concurrent groups G^C in rounds by examining their access integrity. Specifically, the ordering manager assigns a transaction to the current group if all its accesses are the first (i.e., not blocked by other transactions) in the corresponding transaction sets. These assigned transactions are then deleted from transaction sets. Continue the above process until all transactions are assigned. As depicted in Fig. 5(c), TX_4 and TX_7 are assigned to G_1 since their accesses are not blocked by any transactions, while TX_{11} is assigned to G_2 since its access to account $\mathcal{A}_5$ is blocked by TX_7. Due to the paper length limit, we omit the algorithm of this policy here.

C. Parallel Operation of Dual Modes

Among TX^H, there exist transactions that also access cold accounts, termed as TX^{HC}. A few conflicts will occur if the two policies run in parallel. To avoid such conflicts, we give priority to processing TX^{HC} with the incremental state-based processing policy. Then, we adopt the same policy to process the remaining TX^H that only access hotspots and process TX^C using the conflict detection-free processing policy, parallelizing the operation of the two policies. Therefore, such an operation flow of dual modes can fully exploit parallelism while avoiding the above conflicts.

Next, we discuss the theoretical performance improvement brought by our dual-mode mechanism. We assume that the number of transactions to be processed is $\mathbf{TPS}_\Delta$ of transaction appending, and the average time cost to process a single transaction is ψ. The estimated latency for serial processing is $\mathbf{TPS}_\Delta \cdot \psi$. If we use a single concurrent processing policy with maximal x concurrent threads, the estimated processing latency reduces to $\frac{\mathbf{TPS}_\Delta \cdot \psi}{x}$. In contrast, our mechanism parallelizes the operation of dual modes, making the processing of TX^H dominate the overall latency. Meanwhile, we parallelize the execution of **IW** transactions, enabling more available threads to be used, i.e., with a maximal count of x' ($x' > x$). Assuming the proportion of TX^H is α, the estimated processing latency of our mechanism is $\frac{\alpha \cdot \mathbf{TPS}_\Delta \cdot \psi}{x'}$. Compared to the serial and conventional concurrent processing, our dual-mode mechanism potentially achieves performance improvements of $\frac{x'}{\alpha}$ and $\frac{x'}{\alpha \cdot x}$, respectively. The actual processing performance improvement is presented in Section VIII-D.

VI. Security Analysis

In this section, we discuss the security of MorphDAG mainly in terms of how the proposed storage model and transaction processing mechanism guarantee the consistency of blockchain view and state.

A. Safety

Safety means all honest nodes maintain a consistent blockchain view except the last z unconfirmed blocks, i.e., *common-prefix* or *z-consistent*. In our proposed system, concurrent blocks form multiple parallel branches according to the aggregation-based block attachment. We first discuss the convergence of nodes maintaining the same blockchain view for all confirmed blocks on any parallel branch.

Theorem 1. (Convergence of Blockchain View): Consider any block C proposed by an honest node. If it is verified and appended to any parallel branch, all honest nodes will eventually confirm it with a high probability of $1 - \epsilon$.

Proof 1: Assuming that at time t block A is received by all nodes, and another block B has just been proposed. Node j has entered into the next epoch without receiving block B and proposes a new block C' rooted at block A at time t' ($t' - t < \delta$). Since the underlying network is δ-synchronous, block B can be eventually received by all nodes at or before $t + \delta$. Hence, block C' that is only rooted at block A will be regarded as an invalid block. According to the GHOST protocol, once a new block C rooted at blocks A and B is proposed at time $t + \delta$, honest nodes will regard it as a valid block and generate blocks after it. The tie for divergent blockchain views is thus broken, and node j will switch to the valid blockchain view at or before $t + 2\delta$.

Before block C gets confirmed by z blocks, the adversary launches a history-rewrite attack that intends to create a longer malicious branch rooted at blocks A and B. However, we set the safety threshold ω_s for the number of branches, ensuring that the probability of the adversary rewriting the valid history containing block C within z blocks is only ϵ (Section IV-A). Therefore, all honest nodes can confirm block C and maintain the same blockchain view until block C with a probability of $1 - \epsilon$. The proof is thus completed. □

B. Liveness

Like other DAG-based blockchains [7], [9], [10], MorphDAG is also resilient to liveness attacks, where the adversary maintains the tie for multiple divergent blockchain views to prevent honest nodes from achieving a consistent blockchain view. MorphDAG resists liveness attacks by randomly appending newly created blocks to one of the parallel branches. The adversary cannot accurately determine which parallel branch its block will be appended to before block generation. Thus, the adversary cannot

maintain the balance of all branch lengths unless he holds a majority of stake units in the network (contrary to our assumption in Section III-A).

C. State Consistency

The safety property ensures that each node maintains a consistent view for confirmed blocks. Besides, our dual-mode transaction processing mechanism is deterministic, i.e., if nodes input the same set of transactions, the proposed mechanism will process the transactions in a consistent serializable order and output the same state update results. Thus, each node can yield consistent state updates after executing transactions in confirmed blocks. Each node also stores the latest state snapshot for confirmed blocks. Even if an unconfirmed block is subverted, each node will roll back to the previous state snapshot for state consistency.

VII. Implementation

We have implemented a prototype of MorphDAG in Golang. We implement a workload monitor to perceive dynamic workloads, which captures the real-time workload size and the data access pattern via the two-tier transaction pool and the speculative execution, respectively. For the state data storage, we implement a new state database (`MStateDB`) on top of Ethereum [12] that supports incremental state committing. For the on-disk storage of transaction and state data, we use a lightweight key-value storage engine `LevelDB`.[3] We implement a VRF sortition-based PoS consensus protocol and design a parameter analyzer to analyze the degree of storage concurrency and update the sortition target. To support the parallel operation between the dual processing modes, we implement separate ordering managers and thread schedulers for the dual modes, as well as two thread pools for processing transactions accessing hotspot and cold accounts, respectively. For node communication, we implement the P2P network protocol on top of `libp2p`,[4] including node discovery and connection, and employ the *Remote Procedure Call* (RPC) method for communication between MorphDAG clients and servers.

VIII. Evaluation

We evaluate MorphDAG to understand its elasticity under realistic blockchain workloads. Our evaluation aims to answer the following questions: (i) What is the overall performance of MorphDAG under realistic-scale blockchain workloads? (Section VIII-B) (ii) What is the performance gain brought by the elastic degree of storage concurrency theory, and what is the overall performance under large-scale blockchain workloads? (Section VIII-C) (iii) How does the dual-mode transaction processing mechanism perform under synthetic and realistic blockchain workloads? (Section VIII-D)

[3]https://github.com/google/leveldb
[4]https://github.com/libp2p/go-libp2p

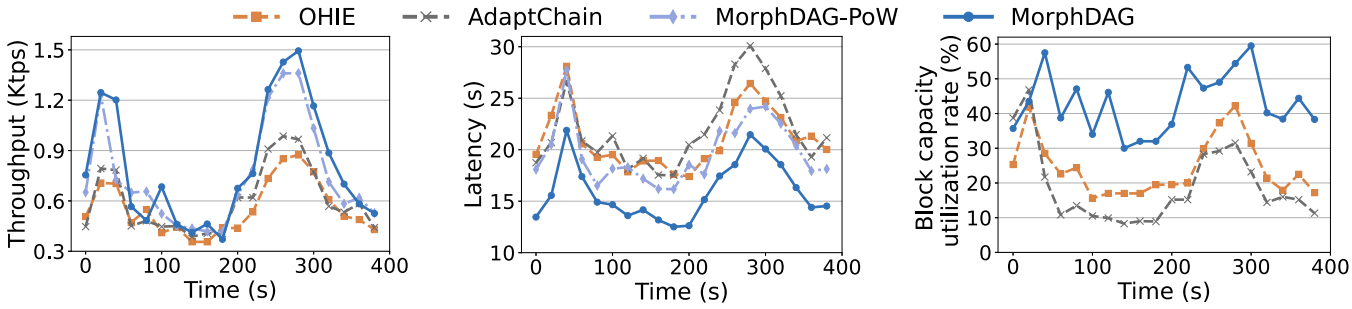
A. Experimental Setup

Testbeds: Our testbed consists of 50 virtual machines (VMs) evenly distributed in five different regions on Alibaba Cloud, i.e., Hong Kong, Tokyo, Virginia, Singapore, and Seoul, which are connected via 200 Mbps network links. Each VM is equipped with a 16-core 3.5-GHz Intel Xeon Platinum 8369B CPU with 32 GB RAM, running Ubuntu 20.04 LTS. By default, we run 100 MorphDAG full nodes on 50 VMs. To simulate the procedure of transaction sending, we run another process in one of the VMs to act as a MorphDAG client to generate and broadcast transactions.

In terms of system parameter settings, we employ the same block generation setting as OHIE [7], i.e., the average interval Δ of each block generation epoch is set as 10 s with the constant multiple $c = 5$. To support efficient parallel propagation of blocks (of average size 800 KB in MorphDAG), each full node uniformly shares the network bandwidth of VMs (200 Mbps), with a maximum bandwidth of 100 Mbps each. We set the constant value ζ_0 that leads to the throughput gain bottleneck as 1 tps. Besides, we set the safety threshold ω_s to 10 via (8) when $\sigma_a = 0.25$ and $z = 30$ such that ϵ is smaller than $4 \cdot 10^{-4}$. We take the same stake distribution setting as Algorand [18], which assigns an equal share of stake to each node.

Workloads: We use two types of blockchain workloads: realistic Ethereum workloads and synthetic workloads. The Ethereum workloads contain a total of 36,142,890 transactions (i.e., 200,000 blocks after block height 14,000,000). We adopt two scales of Ethereum workloads: the realistic-scale and the large-scale with various transaction sending rates, i.e., from 4,000 to 12,000 tps. We generate two synthetic workloads employing the *Smallbank* benchmark (10,000 accounts) with different read-write ratios, i.e., the read-heavy workload (80% read and 20% write) and the balanced workload (50% read and 50% write). We vary the Zipfian distribution to simulate different access skewness. *Baselines:* We compare MorphDAG against two state-of-the-art DAG-based blockchain systems: OHIE [7] with a fixed degree of storage concurrency, and AdaptChain [10] with exponential concurrency adjustment. Since these DAG-based blockchains still adopt PoW-based consensus protocols, we implement a PoW variant of MorphDAG (*MorphDAG-PoW*) for a fair comparison. *MorphDAG-PoW* adopts the elastic degree of storage concurrency theory yet adjusts the PoW difficulty to enable concurrency adjustment. We set OHIE to run 32 and 50 parallel chains under the realistic-scale and the large-scale Ethereum workloads, respectively, while AdaptChain starts from 8 concurrent blocks and expands to a maximum of 100 blocks under all tested workloads. In addition, other system parameters of baselines are consistent with MorphDAG.

To study the impact of our proposed dual-mode transaction processing mechanism (*MorphDAG-Dual*), we compare against two baselines: serial processing that has been adopted in the above two baseline systems, and Nezha [13] which is a concurrent transaction processing scheme towards DAG-based blockchains. We implement the two baselines in MorphDAG, denoted by *MorphDAG-Serial* and *MorphDAG-Nezha*. Furthermore, we implement two variants of *MorphDAG-Dual* for a



(a) Overall throughput (b) Overall latency (c) Block capacity utilization

Fig. 7. Performance comparison of different systems under realistic-scale Ethereum workloads.

detailed comparison: *MorphDAG-Hot* that only employs the incremental state-based processing policy, and *MorphDAG-Cold* that only performs the conflict detection-free processing policy.

B. Performance Under Realistic-Scale Workloads

We compare the overall performance of MorphDAG with OHIE and AdaptChain under the realistic-scale Ethereum workloads shown in Fig. 2(a). We run all experiments for 400 s and evaluate the system performance every 20 s. *Throughput and Latency:* We first measure the effective end-to-end throughput, which removes duplicate transactions. As shown in Fig. 7(a), all evaluated systems exhibit similar performance variation trends under real-time load changes over a runtime of 400 s. However, MorphDAG and *MorphDAG-PoW* present substantially higher throughput compared to AdaptChain and OHIE, especially when the workload size surges. MorphDAG achieves such superiority by adjusting the degree of storage concurrency according to the performance threshold ω_p. Next, we measure the time required for a transaction to be received by full nodes until its updated states are flushed to the disk. Fig. 7(b) shows that the average latency of MorphDAG remains 16 s, which is lower than *MorphDAG-PoW* due to the higher block creation rate of PoS consensus. Moreover, *MorphDAG-PoW* also yields lower system latency than the two baselines, which benefits from our elastic storage and transaction processing.

Block Capacity Utilization Rate: To better understand why MorphDAG has the above advantages in terms of throughput and latency, we compare the average block capacity utilization rate [22], which denotes the ratio of unique transactions in a block to the block size. Note that we only keep one result from MorphDAG and *MorphDAG-PoW* since they adopt the same theoretical value ω_p. As depicted in Fig. 7(c), MorphDAG yields a higher block capacity utilization rate than baselines most of the time. This demonstrates that the performance threshold ω_p can yield high throughput under a relatively low degree of storage concurrency, thereby achieving low resource waste. In contrast, the block capacity utilization rate of AdaptChain exhibits the lowest level of the three due to its exponential concurrency adjustment.

C. Performance Under Large-Scale Workloads

In this set of experiments, we run experiments for 120 s at each transaction sending rate and average the results.

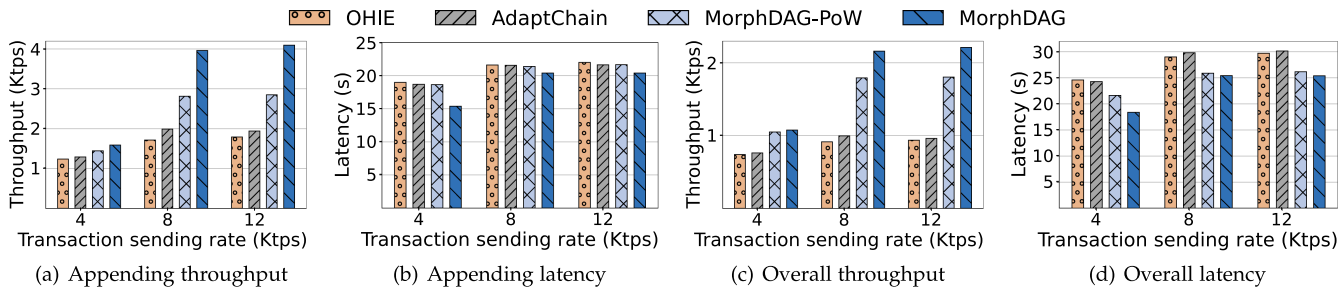
Performance Gain via ω_p: To further explore the performance gain brought by the performance threshold ω_p under large-scale workloads, we measure the system appending throughput and the corresponding latencies. Fig. 8(a) presents that MorphDAG and *MorphDAG-PoW* can reach the peak appending throughput of 4,095 tps and 2,850 tps, respectively. Overall, MorphDAG improves the appending throughput by up to 2.1× and 2.3× over AdaptChain and OHIE, respectively, which demonstrates that ω_p can adapt to large-scale workloads so that most transactions can be appended to the system in time. The trade-off, however, is that the appending latency of MorphDAG is slightly lower than baselines (Fig. 8(b)). The rationale is that many concurrent blocks would start to saturate the network bandwidth, thereby increasing the network propagation latency.

Overall Performance: We further measure the end-to-end overall throughput and latency. Fig. 8(c) shows the significant throughput gains of MorphDAG and *MorphDAG-PoW* over baselines. Specifically, MorphDAG achieves up to 2.3× and 2.4× overall throughput improvement over AdaptChain and OHIE, respectively. As depicted in Fig. 8(d), MorphDAG and *MorphDAG-PoW* exhibit obvious latency degradations than OHIE and AdaptChain due to the performance superiority in transaction processing. To sum up, in addition to the performance gain brought by ω_p, our dual-mode transaction processing mechanism further enhances efficiency under large-scale blockchain workloads.

D. Impact of Dual-Mode Transaction Processing

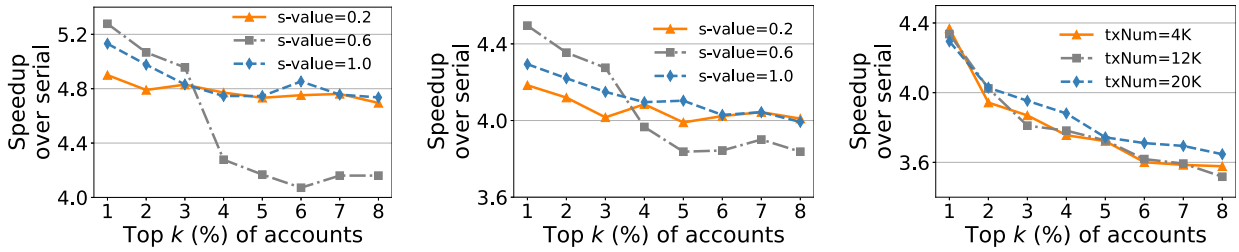
In this set of experiments, we evaluate the transaction processing performance under synthetic workloads at a scale of 8,000 transactions by using three s-values of the Zipfian distribution. Under Ethereum workloads, we evaluate the performance with varying transaction volumes. Results are averaged over ten runs under each workload.

Parameter Calibration: We first study the impact of different k identifying hotspot accounts on transaction processing performance. As depicted in Fig. 9, under *Smallbank* and Ethereum workloads, the speedup performance all exhibit a



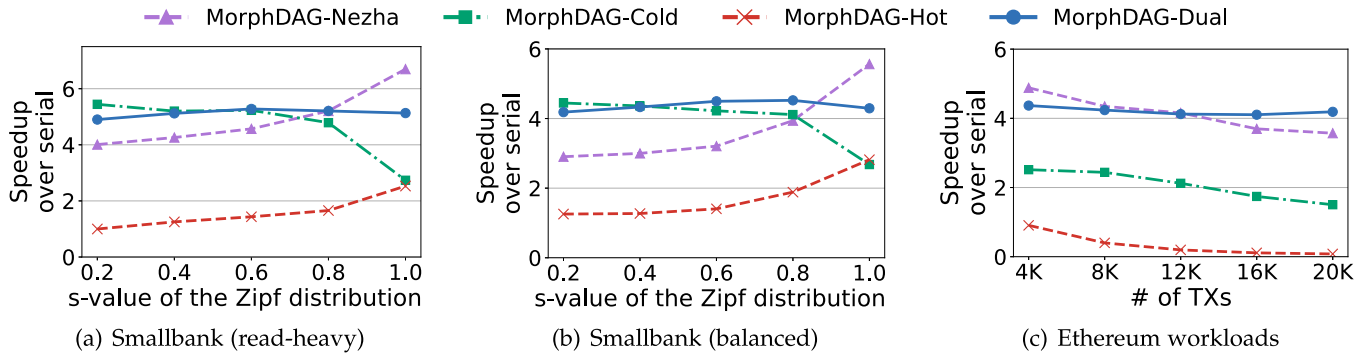
(a) Appending throughput (b) Appending latency (c) Overall throughput (d) Overall latency

Fig. 8. Performance comparison of different systems under large-scale Ethereum workloads.



(a) Smallbank (read-heavy) (b) Smallbank (balanced) (c) Ethereum workloads

Fig. 9. Speedup performance under varying k to identify hotspot accounts.



(a) Smallbank (read-heavy) (b) Smallbank (balanced) (c) Ethereum workloads

Fig. 10. Comparison of speedup performance under synthetic and Ethereum workloads.

trend of descending, which demonstrates that $k = 1$ has the optimal performance among the eight values. Under the uniform account access (s-value $= 0.2$), the performance degradation is slight since the access frequency of hotspots does not change significantly with increasing k. In the case of moderate data access skew (s-value $= 0.6$), there is a significant speedup performance drop between $k = 3$ and $k = 4$. This is because, at $k = 4$, many accounts without conflicting accesses may be identified as hotspots, reducing potential execution concurrency. Moreover, further increasing k does not result in much fluctuation in performance. In contrast, in the case of high data access skew (s-value $= 1.0$), the proportion of accounts with many conflicting accesses may be less than 1%. Therefore, increasing k does not significantly reduce the speedup performance. Considering the above situations, we choose the relatively optimal value of $k = 1$ as the parameter in the next comparison evaluations.

Speedup Performance: We then explore the speedup performance of *MorphDAG-Dual*, *MorphDAG-Nezha*, and the two variants over *MorphDAG-Serial*. Fig. 10(a) and (b) present that, under the read-heavy and balanced workloads, *MorphDAG-Dual* shows better speedup performance than *MorphDAG-Nezha* under the first four s-values, which benefits from the optimization of incremental writes. When the s-value equals 1.0, the speedup performance of *MorphDAG-Nezha* surpasses *MorphDAG-Dual* since *MorphDAG-Nezha* aborts considerable transactions performing writes due to read-write conflicts. As depicted in Fig. 10(c), under Ethereum workloads, the speedup performance of *MorphDAG-Nezha* decreases with the transaction volume rises, instead, *MorphDAG-Dual* shows good scalability. Moreover, both variants of MorphDAG are inferior to *MorphDAG-Dual* under all workloads, demonstrating the advantages of a dual-mode policy over the single one.

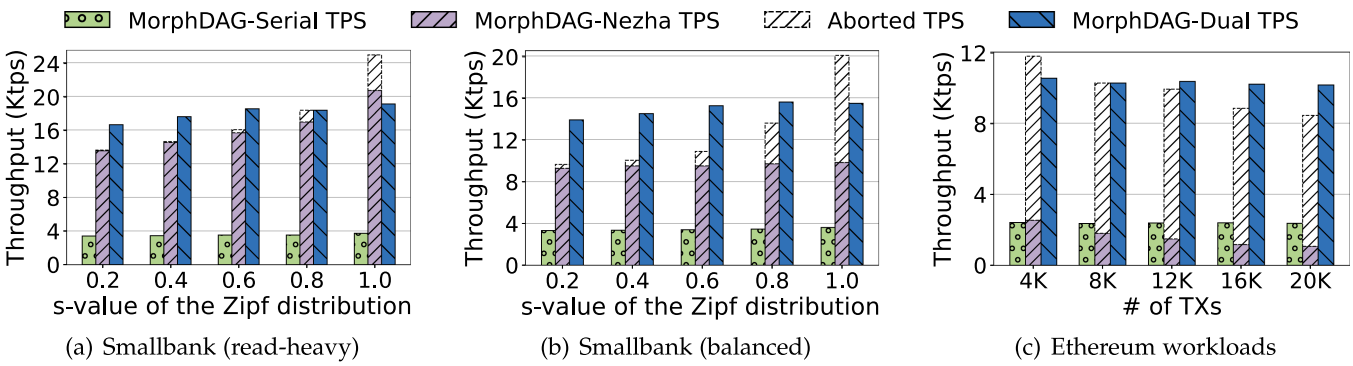


Fig. 11. Comparison of throughput performance under synthetic and Ethereum workloads.

Transaction Processing Throughput: Lastly, we evaluate the effective transaction processing throughput of *MorphDAG-Dual*, *MorphDAG-Nezha*, and *MorphDAG-Serial*. When reads are skewed, the average throughput of *MorphDAG-Dual* is slightly higher than *MorphDAG-Nezha* (Fig. 11(a)). As shown in Fig. 11(b), under the balanced workload, the throughput superiority of *MorphDAG-Dual* over *MorphDAG-Nezha* enlarges. When the account access is highly skewed (s-value = 1.0), the considerable transaction aborts seriously affect the throughput performance of *MorphDAG-Nezha*. Even worse, *MorphDAG-Nezha* yields nearly 84% transaction abort rate under Ethereum workloads (Fig. 11(c)). In comparison, *MorphDAG-Dual* achieves 5.4× and 4.3× throughput improvement over *MorphDAG-Nezha* and *MorphDAG-Serial* on average under Ethereum workloads, respectively.

IX. Related Work

DAG-Based Blockchain Storage: Recent works [7], [8], [9] have adopted a main chain-based or parallel chain-based storage model to improve the throughput of Nakamoto consensus. Some recent efforts are devoted to accelerating the BFT consensus through DAG-based storage models [28], [29], [30]. However, the above works all rely on a fixed degree of storage concurrency, which cannot adapt to dynamic workloads. AdaptChain [10] is the only concurrency adaptive method closest to our work, which provides a non-real-time load perception method that relies on the number of transactions included in previous concurrent blocks. Besides, AdaptChain adopts an exponential adjustment for the degree of storage concurrency yet cannot reach the optimal level appropriate under varying loads.

DAG-Based Blockchain Transaction Processing: Prior DAG-based blockchain systems typically adopt serial transaction processing to trade latency for consistency, which is inefficient for the massive amounts of transactions incurred by many concurrent blocks [16]. Nezha [13] employs the address dependency to enable concurrent transaction processing in DAG-based blockchains yet overlooks the skewed access pattern of realistic blockchain workloads.

Elastic Chain-Based Blockchains: Conventional chain-based blockchains attempt to optimize performance by leveraging resilient BFT consensus [31], [32], adaptive architecture [33], [34], or elastic sharding and storage [25], [35], [36]. For instance, ResilientDB [31] divides nodes into different clusters and pipelines the BFT consensus. FlexChain [33] disaggregates multiple resources into independent resource pools, thus enabling efficient processing of workloads with diverse resource requirements. AdaChain [34] adaptively chooses the optimal blockchain architecture to improve throughput under dynamic workloads. However, the above approaches apply to the permissioned blockchain environment and are limited by their serialized chain structures.

X. Conclusion

MorphDAG realizes an unexplored workload-aware DAG-based blockchain for elastic storage and transaction processing. Based on the elastic degree of storage concurrency theory, MorphDAG achieves an optimal level of throughput under load variations while guaranteeing system security. Further, a dual-mode mechanism is proposed to support efficient transaction processing under skewed data access. We show that MorphDAG outperforms state-of-the-art DAG-based approaches in throughput and latency. Our source code of the MorphDAG prototype will be available at https://github.com/CGCL-codes/MorphDAG.

References

[1] Z. Peng, C. Xu, H. Wang, J. Huang, J. Xu, and X. Chu, "P B-Trace: Privacy-preserving blockchain-based contact tracing to combat pandemics," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2021, pp. 2389–2393.
[2] E. Zhou et al., "MSTDB: A hybrid storage-empowered scalable semantic blockchain database," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 8, pp. 8228–8244, Aug. 2023.
[3] H. Jin and J. Xiao, "Towards trustworthy blockchain systems in the era of "internet of value": Development, challenges, and future trends," *Sci. China Inf. Sci.*, vol. 65, no. 5, pp. 1–11, 2022.
[4] S. Popov, "The tangle," 2018. [Online]. Available: http://cryptoverze.s3.us-east-2.amazonaws.com/wp-content/uploads/2018/11/10012054/IOTA-MIOTA-Whitepaper.pdf
[5] A. Churyumov, "Byteball: A decentralized system for storage and transfer of value," 2016. [Online]. Available: https://byteball.org/Byteball.pdf
[6] C. LeMahieu, "Nano: A feeless distributed cryptocurrency network," 2018. [Online]. Available: https://nano.org/en/whitepaper
[7] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "OHIE: Blockchain scaling made simple," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 90–105.
[8] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 585–602.

[9] C. Li et al., "A decentralized blockchain with high throughput and fast confirmation," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 515–528.
[10] J. Xu, Q. Xie, S. Peng, C. Wang, and X. Jia, "AdaptChain: Adaptive scaling blockchain with transaction deduplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 6, pp. 1909–1922, Jun. 2023.
[11] J. Ni, J. Xiao, S. Zhang, B. Li, B. Li, and H. Jin, "FLUID: Towards efficient continuous transaction processing in DAG-based blockchains," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12679–12692, Dec. 2023.
[12] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf
[13] J. Xiao et al., "Nezha: Exploiting concurrency for transaction processing in DAG-based blockchains," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 269–279.
[14] T. Jiang, G. Zhang, Z. Li, and W. Zheng, "Aurogon: Taming aborts in all phases for distributed in-memory transactions," in *Proc. USENIX Conf. File Storage Technol.*, 2022, pp. 217–232.
[15] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 543–557.
[16] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 11, pp. 5070–5083, Nov. 2022.
[17] Z. Chen, X. Qi, X. Du, Z. Zhang, and C. Jin, "PEEP: A parallel execution engine for permissioned blockchain systems," in *Proc. 26th Database Syst. Adv. Appl.*, 2021, pp. 341–357.
[18] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proc. ACM Symp. Operating Syst. Princ.*, 2017, pp. 51–68.
[19] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2019, pp. 23–41.
[20] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[21] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Proc. IEEE Annu. Symp. Found. Comput. Sci.*, 1999, pp. 120–130.
[22] C. Chen, X. Chen, and Z. Fang, "TIPS: Transaction inclusion protocol with signaling in DAG-based blockchain," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 12, pp. 3685–3701, Dec. 2022.
[23] R. Taft et al., "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," in *Proc. Int. Conf. Very Large Data Bases*, 2015, pp. 245–256.
[24] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2015, pp. 507–527.
[25] H. Huang et al., "Elastic resource allocation against imbalanced transaction assignments in sharding-based permissioned blockchains," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 10, pp. 2372–2385, Oct. 2022.
[26] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *Proc. IEEE P2P*, 2013, pp. 1–10.
[27] C. Grunspan and R. Pérez-Marco, "Double spend races," *Int. J. Theor. Appl. Financial*, vol. 21, no. 08, pp. 1–32, 2018.
[28] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Proc. Annu. ACM Symp. Princ. Distrib. Comput.*, 2021, pp. 165–175.
[29] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 2705–2718.
[30] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A DAG-based mempool and efficient BFT consensus," in *Proc. Eur. Conf. Comput. Syst.*, 2022, pp. 34–50.
[31] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "ResilientDB: Global scale resilient blockchain fabric," in *Proc. Int. Conf. Very Large Data Bases*, 2020, pp. 868–883.
[32] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: Resilient concurrent consensus for high-throughput secure transaction processing," in *Proc. IEEE Int. Conf. Data. Eng.*, 2021, pp. 1392–1403.
[33] C. Wu, M. J. Amiri, J. Asch, H. Nagda, Q. Zhang, and B. T. Loo, "FlexChain: An elastic disaggregated blockchain," in *Proc. Int. Conf. Very Large Data Bases*, 2022, pp. 23–36.
[34] C. Wu, B. Mehta, M. J. Amiri, R. Marcus, and B. T. Loo, "AdaChain: A learned adaptive blockchain," in *Proc. Int. Conf. Very Large Data Bases*, 2023, pp. 2033–2046.
[35] X. Qi, Z. Zhang, C. Jin, and A. Zhou, "A reliable storage partition for permissioned blockchain," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 1, pp. 14–27, Jan. 2021.
[36] Z. Du, H.-F. Qian, and X. Pang, "PartitionChain: A scalable and reliable data storage strategy for permissioned blockchain," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 4, pp. 4124–4136, Apr. 2023.

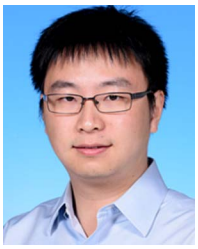
Shijie Zhang (Student Member, IEEE) received the MS degree from the Department of Software, Sangmyung University, Republic of Korea, in 2019. He is currently working toward the PhD degree with the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), supervised by Prof. Jiang Xiao. His current research mainly interests include blockchain and distributed systems.

Jiang Xiao (Member, IEEE) received the BSc degree from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2009, and the PhD degree from the Hong Kong University of Science and Technology (HKUST), in 2014. She is currently a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. Her research interests include blockchain, and distributed computing. Her awards include CCF-Intel Young Faculty Research Program 2017, Hubei Downlight Program 2018, ACM Wuhan Rising Star Award 2019, Knowledge Innovation Program of Wuhan-Shuguang 2022, and Best Paper Awards from IEEE ICPADS/GLOBECOM/GPC/BLOCKCHAIN.

Enping Wu (Student Member, IEEE) received the bachelor's degree from Donghua University (DHU), Shanghai, China, in 2019. He is currently working toward the master's degree supervised by Prof. Jiang Xiao. His research interests mainly include blockchain and distributed systems.

Feng Cheng (Student Member, IEEE) received the bachelor's degree from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2021. She is currently working toward the master's degree supervised by Prof. Jiang Xiao. Her research interests mainly include blockchain and distributed systems.

Bo Li (Fellow, IEEE) received the BEng (summa cum laude) degree in computer science from Tsinghua University, Beijing, China, and the PhD degree from ECE Department, University of Massachusetts at Amherst. He is a chair professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He was a Cheung Kong Scholar visiting chair professor with Shanghai Jiao Tong University (2010–2016), and was the chief technical advisor for ChinaCache Corp. (NASDAQ:CCIH), a leading CDN provider. He made pioneering contributions in multimedia communications and the Internet video broadcast, which attracted significant investment from industry and received the Test-of-Time Best Paper Award from IEEE INFOCOM (2015). He received six Best Paper Awards from IEEE including INFOCOM (2021). He was the Co-TPC chair for IEEE INFOCOM (2004).



Wei Wang (Member, IEEE) received the BEng and MEng degrees from the Department of Electrical Engineering, Shanghai Jiao Tong University, China, in 2007 and 2010, respectively, and the PhD degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2015. Since 2015, he has been with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST), where he is currently an associate professor. He is also affiliated with the Big Data Institute, HKUST. His research interests include distributed systems, with focus on serverless computing, machine learning systems, and cloud resource management. He has published extensively in the premier conferences and journals of his field. His research has won the Best Paper Runner Up Awards of IEEE ICDCS 2021 and USENIX ICAC 2013.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 1994. He is a chair professor of computer science and engineering with the Huazhong University of Science and Technology in China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with the University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is a fellow of the CCF, and a life member of the ACM. He has co-authored more than 20 books and published more than 900 research papers. His research interests include computer architecture, parallel and distributed computing, Big Data processing, data storage, and system security.