2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3315-1596-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICPADS63350.2024.00087

BachLedger: Orchestrating Parallel Execution with Dynamic Dependency Detection and Seamless Scheduling

Yi Yang[1], Guangyong Shang[2], Guangpeng Qi[2], Zhen Ma[2], Yaxiong Liu[2], Jiazhou Tian[1], Aocheng Duan[3], Meng Zhang[2], Jingying Li[2], Xuan Ding[1]

[1]School of Software, Tsinghua University, Beijing, China
[2]Inspur Yunzhou Industrial Internet Co., Ltd, Jinan, Shandong, China
[3]School of Cyber Engineering, Xidian University, Xi'an, Shaanxi, China
yangyi23@mails.tsinghua.edu.cn
{shangguangyong, qigp, mazhenrj, liuyaxiong01}@inspur.com
duanac@stu.xidian.edu.cn
tjz22@mails.tsinghua.edu.cn
{zhangm_lc, lijingying01}@inspur.com
dingxuan@tsinghua.edu.cn

***Abstract*—Blockchain technology inherently necessitates redundant computation to achieve consensus among untrusted parties because of its fundamental threat model. This requirement, however, compromises system performance and impedes the widespread adoption of blockchain. To leverage existing physical resources, current research on high-performance consortium blockchain algorithms and architectures frequently employs cluster-node architectures to expand the parallel processing capability of traditional single physical nodes. Our investigation reveals a significant trend as the parallel capability of individual nodes improves. The idle time caused by synchronization of all transactions within each block, previously considered negligible, has become increasingly significant. To address this, we present BachLedger, which implements Seamless Scheduling to fully utilize inter-block thread idle time, thereby augmenting system resource utilization and achieving overall performance improvements. Our experimental results demonstrate that our algorithm surpasses current state-of-the-art (SOTA) performance levels in high-performance consortium blockchains and effectively resolves the aforementioned synchronization issue. Furthermore, this scheduling algorithm offers enhanced scalability for BachLedger, positioning it as a promising solution for future blockchain implementations.**

***Index Terms*—blockchain, parallel execution, dependency detection, scheduling, high-performance computing**

I. Introduction

Blockchain systems, as a decentralized distributed computing paradigm, face unique challenges in performance and scalability compared to traditional distributed systems like [1]–[3]. Although both adopt distributed architectures, the fundamental difference in their underlying threat models directly leads to a vast divergence in algorithm design, which in turn results in a significant performance gap in the overall processing pipeline comprised of consensus, execution, and storage.

In traditional distributed systems, nodes are typically treated as trustworthy, allowing them to leverage work partitioning and load balancing to maximize system throughput. Blockchain systems operate under a totally distinct threat model where nodes inherently distrust each other. Every node must independently validate the same set of transactions to achieve Byzantine fault tolerance, essentially performing redundant computations. This consensus requirement significantly hinders blockchain's execution parallelism and scalability compared to its traditional counterparts.

Cluster-node architectures, as exemplified in [4]–[10], represent a significant trend in blockchain technology. By operating a cluster of servers as a single logical node, with all nodes within the cluster maintaining mutual trust, blockchain systems can achieve higher performance comparable to traditional distributed systems while preserving their original security assumptions. However, in the context of enhanced parallelism within a single logical node, previously inconsequential inefficiencies may manifest as significant performance bottlenecks, potentially impeding the overall system throughput.

In prevailing blockchain architectures, the initiation of transaction execution within a given block is typically dependent upon the complete finalization of antecedent blocks. The synchronization of all transactions before completing formation often leads to inefficient use of computing resources. To simplify the analysis, assume that the execution time of a single transaction is a random variable X with an expected value μ and variance σ^2, and the number of transactions assigned to each thread is approximately a constant M. According to the Central Limit Theorem, the relative difference between two independent threads p and q can be defined as

Project supported by the Key Program of the National Natural Science Foundation of China (Grant No. 62232004).
Xuan Ding is the corresponding author.

979-8-3315-1596-6/24/$31.00 ©2024 IEEE
DOI 10.1109/ICPADS63350.2024.00087

$\frac{Y_p - Y_q}{M\mu} \sim N\left(0, \frac{2\sigma^2}{M\mu^2}\right)$, which converges in probability to 0 when $M \to +\infty$. This implies that when the number of available threads in a single node is limited and the number of transactions assigned to each thread (i.e., M) is sufficiently large, there is minimal inter-thread waiting time, resulting in negligible idle time. However, in a clustered single-node environment with an increased number of available threads per node, this assumption no longer holds, necessitating the optimization of idle time to improve overall system efficiency. Figure 1 illustrates this point through a simple example.

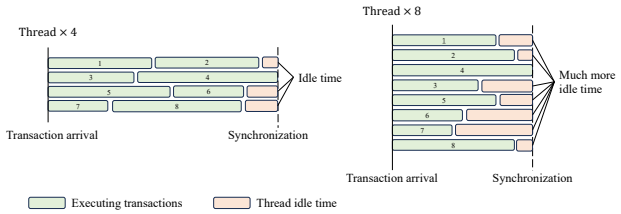

Fig. 1. Illustration of the increasing impact of idle time with more available threads in transaction execution.

To corroborate the aforementioned theoretical derivation, we constructed a blockchain system framework implementing a conventional inter-block parallel algorithm incorporating seams. We kept N constant (defaulting to 100 transactions) and gradually increased the number of available threads to observe the variation in the proportion of idle time to the overall time. We define the total execution time T_i of each transaction t_i within a block as the duration including the last effective execution time and all previous ineffective execution times caused by conflicts with other transactions. The time taken for a block to complete all transactions effectively is denoted as T. Therefore, the relative thread idle time is defined as $\delta = 1 - \frac{1}{NT}\left(\sum_{i=1}^{N} T_i\right)$. As illustrated in Figure 2, with the increase in available threads, δ gradually converges to a higher level, while its mean and median values demonstrate an upward trend. This indicates that our optimization is highly necessary.

The synchronization of all transactions before block execution completion, while prevalent, primarily stems from implementation convenience rather than algorithmic necessity. High-parallelism blockchain systems essentially aim to achieve "dependency preposition". For instance, FISCO-BCOS[5] and SChain[4] demonstrate that execution results (i.e., read-write sets) can be omitted before ordering. They define "transactions included in the previous block" as their inter-block dependency and utilize deterministic scheduling to ensure result consistency. By positioning consensus at the pipeline's inception, these systems enable earlier commencement of the next block's processing, achieving high inter-block parallelism.

These methods, however, introduce another pipeline dependency: the necessity for complete execution of all transactions within a block prior to initiating the subsequent block's execution, owing to each block's reliance on the state snapshot generated upon its predecessor's completion. This dependency,

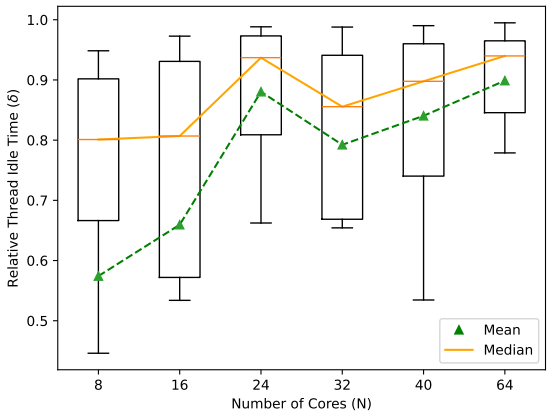

Fig. 2. Idle Time Impact with Increased Parallel Threads.

however, is not absolute: subsequent block transactions may not necessarily conflict with those within the current block. During the synchronization of transactions within a block, as illustrated in Figure 1, idle threads have the potential to optimistically execute transactions from the succeeding block. Consequently, we propose a Seamless Scheduling algorithm to capitalize on this opportunity.

Our research reveals that transaction scheduling within a block invariably maintains a Directed Acyclic Graph (DAG) structure, either explicitly or implicitly, while identifying nodes with zero in-degree in the DAG. Systems like Schain[4] and Monad[11] employ static analysis to extract transaction DAGs and dispatch transactions accordingly. In contrast, other systems such as [7], [10], [12], [13] identify transaction dependencies as conflicts arise during execution. In Block-STM[13], the multi-versioned data structure effectively represents a series of directed edges in DAGs.

In this paper, we introduces BachLedger, a high-performance blockchain, featuring Seamless Scheduling to optimize idle time between blocks. Our algorithm draws inspiration from the structure of Bach's fugue scores. Batches of transactions are individually encapsulated as blocks while being scheduled in a single queue to maximize overall efficiency. Prior to the complete formation of a block, transactions in subsequent blocks can be executed in advance if idle threads are available.

In order to facilitate the scheduling of cross-block transactions within a single queue, we have devised a deterministic sequence number incorporating a semantic prefix as transactions' priority code. This innovative approach enables the uniform processing of priority codes across disparate blocks. Our approach eschews the implementation of a centralized component, instead leveraging hash values to establish transaction priorities. The inherent certainty and unpredictability of the hash function render it an ideal mechanism for achieving consensus on transaction priorities across a cluster, without

necessitating complex consensus protocols.

We have introduced a novel data structure, termed the "Ownership Table", which maintains a record of the transactions that possess ownership of specific data entries in storage. During each iteration, only those transactions that have exclusive ownership of all entries they intend to read or write are eligible for commitment. In essence, this table catalogues all nodes with zero in-degree within an implicit DAG at each iteration. Our algorithm is designed to extract all such transactions from an expanding queue in topological order.

In summary, our research presents the following key contributions:

- **Seamless Scheduling for Cluster-Node Job Dispatching:** We demonstrate the critical importance of utilizing idle computing resources between block processing procedures. Our novel protocol enables Seamless Scheduling of cross-block transactions within a unified queue, maximizing resource utilization.
- **Ownership Table and Priority Codes for Dependency Management:** We introduce the Ownership Table, a sophisticated data structure, coupled with semantically prefixed sequence numbers as priority code. This innovation facilitates dynamic conflict detection and resolution among transactions, while concurrently constructing an implicit DAG. The Ownership Table efficiently records nodes with zero in-degree within this DAG.
- **High-Performance Blockchain with Cluster-Node Architecture:** Leveraging these advancements, we have developed BachLedger, a high-performance blockchain system. Its cluster-node architecture significantly enhances parallelism capabilities. Our Seamless Scheduling mechanism endows BachLedger with substantial performance advantages over existing solutions.

II. Related Work

In the realm of blockchain architecture, enhancing performance predominantly involves augmenting system parallelism, given the constraint of utilizing extant hardware infrastructure. Parallelism in this context can be conceptualized along two primary dimensions: firstly, the development of a distributed physical architecture to amplify the system's capacity for parallel processing; and secondly, the optimization of algorithms to incorporate more independent subroutines amenable to concurrent execution.

A. Inter-block Parallelism

The majority of traditional blockchain systems, particularly those utilizing Proof of Work (PoW) consensus mechanisms, implement an Ordering-Execution-validation (OEV) architecture. This architecture follows a three-step transaction processing paradigm: (1) transaction aggregation for block formation; (2) network-wide consensus on the composition of the subsequent block; and (3) transaction execution and state update by individual nodes after validation. To maintain consistency across the distributed ledger, the execution scheduling in the final step necessitates determinism. The most rudimentary approach involves delegating transaction execution to a single primary node, selected via the consensus protocol, or employing strictly sequential execution.

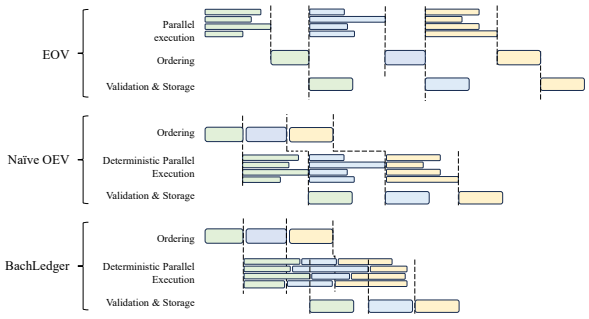

Fig. 3. Comparison of OEV and EOV architecture.

In contrast, Fabric[7] introduces an execution-ordering-validation (EOV) architecture, aiming to facilitate parallel execution sequencing and systems like [6], [14] follow this architecture. As shown in Figure 3, EOV architecture cannot support high pipeline parallelism. Although systems like Fabric adapt optimistic execution, delayed storage snapshot updates result in high abortion rate. Nevertheless, the permissioned blockchains with OEV architecture including [4], [5], [12] demonstrates superior potential for enhanced inter-block parallelism, effectively enabling pipeline parallelism. Furthermore, recent advancements in deterministic parallel scheduling algorithms have significantly bolstered the overall parallelism capabilities of the OEV architecture, rendering it increasingly advantageous in terms of performance and scalability.

B. Critical Dependency Prepositioning

"Critical dependency" refers to essential information from a preceding block or transaction necessary for determining its successor, whose determination lies on the critical path of the parallel workflow. This necessitates sequential generation of dependent information, such as block hash values or previous block execution results (snapshots). Critical dependencies are not absolute but rather algorithm-dependent. The key to enhancing inter-block parallelism in OEV architecture is "critical dependency prepositioning". This involves designing algorithms to determine critical dependency information earlier and eliminate unnecessary dependencies, thereby enabling earlier initiation of dependent processes.

Ordering, as the initial phase of OEV architecture, inevitably lies on the critical path. Blockchains employing the OEV architecture, such as those presented in [4], [5], [12], enhance inter-block parallelism by prioritizing the ordering phase at the outset of their processing pipeline. DispersedLedger[15] posits that consensus primarily ensures data consistency among participants, allowing for asynchronous content retrieval. Bidl isolates transaction ordering as the crucial consensus role,

validating it through a parallel consensus process. The system proposed by [16], designed for specific requirements, relaxes the constraint of maintaining absolute consistency in block ordering. These approaches fundamentally seek to eliminate superfluous critical dependencies while strategically prepositioning inevitable dependencies, thus enhancing parallelism.

C. Deterministic Scheduling

Deterministic scheduling must ensure deterministic serializability, wherein the effect of a transaction schedule is consistently equivalent to a specific serialized execution sequence. Two primary approaches exist: (1) utilizing static analyzers, such as Slither[17], to generate a Directed Acyclic Graph (DAG) a priori, and (2) dynamically detecting inter-transaction dependencies upon conflict occurrence. Monad[11] employs static dependency analysis to construct a DAG for parallel execution, reverting to serialized execution when static analysis proves insufficient. Similarly, SChain[4] requires a pre-generated DAG to coordinate transactions across disparate execution shards.

The dynamic approach essentially maintains an implicit Directed Acyclic Graph (DAG) through alternative data structures. Systems proposed in [7], [13] employ tables containing versioned read-write results, effectively recording the directed edges of an implicit DAG. These algorithms iteratively identify nodes with zero in-degree within the DAG. Given the potential unreliability of static analysis and the necessity for serialized execution fallbacks in static-analysis-based schedules, deterministic scheduling has emerged as an increasingly preferable option in numerous scenarios.

D. Sharding

Sharding is essential for blockchain scalability. SlimChain enhances throughput by partitioning transactions for parallel processing, reducing computational load [18], while Algorand uses Verifiable Random Functions (VRF) to improve consensus efficiency [19]. SlimChain focuses on transaction optimization, and Algorand on consensus.

Blockumulus brings sharding to cloud computing, enabling scalable decentralized smart contracts across transaction throughput, storage, and computation [20]. Its overlay consensus uses cloud resources while maintaining decentralization. These approaches showcase sharding's flexibility in improving blockchain performance. The scheduling algorithm proposed in this paper can be applied to execution-layer sharding for task allocation and conflict resolution.

III. System Design

A. Prerequisite and Assumption

The security model of our system is predicated on a distributed network of nodes deployed by multiple distinct organizations. Each organization maintains a cluster node, with intra-cluster nodes operating under a trust relationship, while inter-cluster interactions are characterized by mutual distrust. Throughout this discourse, the term "node" implicitly refers to a "cluster node", unless explicitly qualified as an "execution node", "ordering node", or other functionally specific sub-node within a cluster. A fundamental assumption underpinning our system's integrity is that the proportion of malicious or faulty nodes never exceeds one-third of the total node population. Formally, given a total of n cluster nodes, where $n > 3f + 1$, the number of Byzantine cluster nodes is strictly less than f. This assumption is crucial for the Byzantine Fault Tolerant (BFT) consensus algorithm employed in our consensus phase. Furthermore, the network topology is modeled as partially synchronous, wherein no theoretical upper bound on message delivery time exists. However, the model incorporates a Global Stabilization Time (GST), ensuring that system synchrony is inevitably restored post-GST in the event of temporary network perturbations or blockages.

B. BachLedger's Workflow

BachLedger is a blockchain system that employs the Optimistic Execution and Validation (OEV) algorithmic architecture. Its operational workflow comprises three primary phases: Ordering, Execution, Validation and then Storage as shown in Figure 4.

a) Ordering Phase: Ordering phase initiates with transactions being directed to a transaction pool, which facilitates transaction collection and flooding broadcast. Block creation is triggered either when the pool accumulates sufficient transactions or at predetermined intervals (default: 50 milliseconds) if the pool is non-empty. BachLedger utilizes TBFT as its consensus algorithm. Upon receiving a packaging signal, the consensus-elected primary node constructs a block and initiates network-wide consensus using TBFT. This process ensures network-wide agreement on a block containing an identical set of transactions. Consensually approved transactions are then queued for scheduling.

b) Execution Phase: Each queued transaction receives a distinctive identifier that determines its execution priority. This identifier consists of two parts: a semantic prefix and a hash-derived value. The hash component ensures uniqueness, while the prefix maintains the order of cross-block transactions and tracks whether the data items owned by each transaction have been released. The scheduling algorithm leverages an Ownership Table to detect and manage conflicts, guaranteeing that conflicting transactions are confirmed in order of their identifier-based priority.

c) Validation and Storage Phase: The Validation phase addresses potential non-compliance in transaction execution across nodes. Post-execution, nodes broadcast signatures and block hashes containing transaction results. Each node compares received block hashes with its local blocks, retaining signatures for matching blocks. A block requires signatures from over two-thirds of the nodes for validation. While this introduces a dependency, it is not on the critical path and does not bottleneck the system, as it does not impede subsequent pipeline processes.

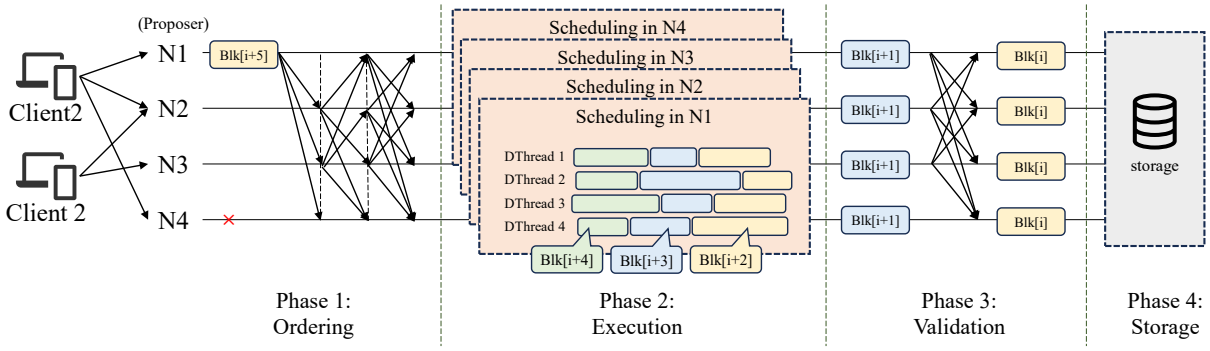

Fig. 4. The overall workflow of BachLedger.

IV. Seamless Scheduling Protocol

A. Ownership Table and Semantically-prefixed Priority Code

In this section, we elucidate the intricacies of the Seamless Scheduling protocol, which constitutes the core of our system design. The fundamental principle underlying this protocol is the iterative identification and collection of nodes with zero in-degree within the DAG. To facilitate this process, we have devised an Ownership Table that serves as a comprehensive record of these zero in-degree nodes. Through dependency detection in each iteration, these nodes are systematically identified and confirmed. The Ownership Table is strategically designed to enable the system to handle this logical process in a uniform and efficient manner.

Algorithm 1 Ownership Entry's Methods

```
1:  const DISOWNED ← 1
2:  const OWNED ← 0
3:  procedure ReleaseOwnership(self)
4:      self.mutex.Lock()
5:      defer self.mutex.Unlock()
6:      self.owner[0] ← DISOWNED
7:  procedure CheckOwnership(self, who)
8:      self.mutex.RLock()
9:      defer self.mutex.RUnlock()
10:     return who ≤ self.owner
11: procedure TrySetOwner(self, who)
12:     if not self.CheckOwnership(who) then
13:         return false
14:     self.mutex.Lock()
15:     defer self.mutex.Unlock()
16:     if who ≤ self.owner then
17:         self.owner ← who
18:         return true
19:     return false
```

The Ownership Table is conceptualized as a mapping structure wherein each storage key corresponds to its current owner's priority code. To maximize parallelism, we implement fine-grained read-write locks at the ownership entry level. Exclusive locks are used only for ownership modifications of specific storage keys, while shared locks suffice for reads. This ensures operations on one key's ownership do not impede access to others. The priority code, consisting of a sequence number with a semantic prefix, is comprised of three distinct components: (1) a single bit indicator denoting the release status of the ownership; (2) the transaction's block height; and (3) a hash-derived value computed from both the transaction itself and the encompassing block of transactions. Since adversaries cannot predict other transactions in a block, they cannot craft a hash value that is smaller than the others within the same block.

In our design paradigm, the lowest numerical priority code is accorded the highest execution priority. Therefore, this structure ensures several key properties: firstly, the uniqueness of priority codes is guaranteed by the third field; secondly, transactions within the earliest block are prioritized for execution due to the second field. What's more, when transactions are confirmed and the release of ownership is necessitated, we employ a simple yet effective mechanism: the first bit of the corresponding record in the table is set to 1. This operation ensures that the released ownership's priority code becomes numerically larger than any other code currently in the waiting queue, effectively deprioritizing it. The priority mechanism and fine-grained mutex in the implementation of the ownership entry are illustrated in Algorithm 1.

B. Seamless Scheduling Design

After all transactions are packaged into blocks and reach consensus through the ordering stage, they are disassembled and merged into a common waiting queue. Specifically, transactions from different blocks are sorted using the same sequential rule, known as the semantically-prefixed priority code mentioned earlier. As illustrated in the Figure 5 and the Algorithm 2, each arriving transaction is optimistically executed in advance. Within the critical section, conflicts among the transactions of the earliest unconfirmed block are processed. When a transaction conflict is detected, the lower-

priority transaction is re-executed until the last transaction in the block is completed. At this point, transactions from the next block can enter the critical section.

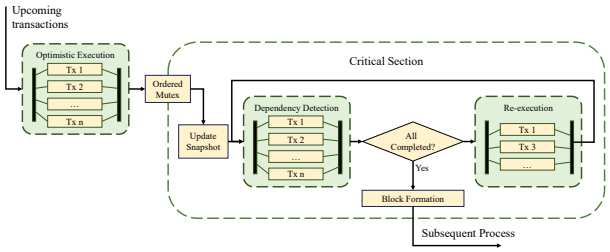

Fig. 5. A detailed illustration of Seamless Scheduling.

In both the optimistic execution and re-execution phases, transactions are processed similarly, except for the attachment of the priority code during optimistic execution. In these two phases, a coordinator dispatches transactions to threads in different executors for parallel execution and synchronizes to obtain all the complete read-write sets. The coordinator iterates through the entire write set to attempt to acquire the corresponding ownership. During the conflict detection phase, the read-write set of each transaction from the previous execution phase is rechecked to verify whether the transaction holds all the storage keys it needs to read and write by comparing their priority codes with the owner fields in the entry. Transactions that fail the conflict check, i.e., those that do not obtain ownership of all necessary storage keys, are added to a list for re-execution in the next round.

V. Experiment

Our system primarily aims to address the impact on resource efficiency as the number of parallel threads increases. Additionally, we have adopted the OEV architecture, as our research indicates its capacity to achieve higher pipeline parallelism. To validate our theoretical derivations, assess end-to-end performance, and evaluate system scalability, we have conducted a comprehensive series of experiments. Through these investigations, we seek to answer the following research questions:

- Does our theoretical derivation hold true: as the number of available threads increases, does the proportion of idle time also increase?
- Can our proposed BachLedger using Seamless Scheduling solve the fragmented time issue derived from our theoretical analysis?
- Can our proposed BachLedger achieve and surpass the SOTA level in overall performance as the number of available threads increases?

A. Experiment Setup

a) Baseline: As previously elucidated, we have opted for the OEV architecture due to its inherent potential for enhanced parallelism. BachLedger further augments this parallelism through the implementation of Seamless Scheduling

Algorithm 2 Seamless Scheduling

Input: Transactions T in a block
Output: Processed Transactions
procedure SCHEDULE(T)
  $snapshot \leftarrow$ new snapshot
  $Q_s \leftarrow$ new queue ▷ *a queue of approved transactions*
  $Q_e \leftarrow$ optimisticExecuteTxs($block, snapshot$)
  mu.Lock()
  defer mu.Unlock()
  updateSnapshot()
  while len(Q_e) > 0 **do**
    $Q_a \leftarrow$ detectConflict($Q_e, Q_s, snapshot$)
    $Q_e \leftarrow$ reExecuteTxs($block, Q_a, snapshot, Q_s$)
  $blk \leftarrow$ formalizeBlock(Q_s)
  return blk
function OPTIMISTICEXECUTETXS(block, snapshot)
  $Q_e \leftarrow$ new queue ▷ *a queue of executed transactions*
  for all $tx \in block.txs$ **parallel do**
    $hashField \leftarrow$ hash(tx, hash($block.txs$))
    $tx.priority \leftarrow$ join($0, block.height, hashFiled$)
    $tx.rset, tx.wset \leftarrow$ execute($tx, snapshot$)
    for all $w \in tx.wset$ **do**
      table[w.key].TrySetOwner(tx.priority)
    append(Q_e, tx)
  Synchronize with all threads
  return Q_e
function DETECTCONFLICT(Q_e, Q_s, snapshot)
  $Q_a \leftarrow$ new queue ▷ *a queue of aborted transactions*
  for all $tx \in Q_e$ **parallel do**
    handleConflictDetection(tx, Q_a, Q_s)
  Synchronize with all threads
  return Q_a
function HANDLECONFLICTDETECTION(tx, Q_a, Q_s)
  for all $w \in tx.wset$ **do**
    if $table[w.key]$.CheckOwnership($tx.priority$) = $false$ **then**
      append(Q_a, tx)
      return
  for all $r \in tx.rset$ **do**
    if $table[r.key]$.CheckOwnership($tx.priority$) = $false$ **then**
      append(Q_a, tx)
      return
  append(Q_s, tx)
function REEXECUTETXS(block, Q_a, snapshot, Q_s)
  $Q_e \leftarrow$ new queue
  for all $tx \in Q_a$ **parallel do**
    $tx.rset, tx.wset \leftarrow$ execute($tx, snapshot$)
    for all $w \in tx.wset$ **do**
      table[w.key].TrySetOwner(tx.priority)
    append(Q_e, tx)
  Synchronize with all threads
  return Q_e

in its execution phase. To rigorously evaluate the efficacy of our approach, we conduct a comprehensive comparative analysis. Specifically, we benchmark BachLedger against two prominent blockchain systems:

- ChainMaker [14]: A widely adopted consortium blockchain leveraging the EOV architecture.
- FISCO-BCOS [5]: A state-of-the-art consortium blockchain solution, exemplifying the OEV architecture. In this study, we utilize its lightweight Air version.

This comparative framework allows us to assess the performance improvements achieved by BachLedger, not only against a different architectural approach (EOV) but also against a leading implementation within the same architectural category (OEV). Through this analysis, we aim to demonstrate the superior parallelism and efficiency of BachLedger in relation to both established and cutting-edge blockchain systems.

b) Physical Environment and Deployment: This study utilized an Inspur NF5280M5 server, featuring dual Intel Xeon Gold 5218 processors (2.30 GHz base frequency). The system leverages Hyper-Threading technology to provide 64 logical cores and incorporates 64 GB of RAM. In the context of this study, four ordering nodes are deployed as a standard configuration. To rigorously assess the impact of thread availability on the performance of execution nodes, we strategically manipulate the number of accessible cores on the server using Linux Control Groups. This method enables precise and consistent control over resource allocation across different framework designs, allowing for a detailed analysis of scalability and efficiency in managing varying computational loads.

c) Workload: In our experiments, we utilized the ERC-20 token transfer scenario as the primary workload, involving transactions that require simultaneous read and write operations on identical accounts. We initialized 100 accounts, each with sufficient approved tokens, and generated a randomized set of transfers among these accounts. The stochastic nature of the transaction sequence introduces random conflicts, resulting in variable lifecycle durations for each transaction from initial execution to final conflict-free execution. Consequently, heterogeneous execution times generate fragmented thread idle time between blocks. This simulation methodology effectively evaluates our proposed Seamless Scheduling mechanism.

B. End-to-end Performance Experience

As illustrated in Figure 6, we conducted comparative experiments using ChainMaker, FISCO-BCOS, and BachLedger under controlled conditions. The experiments were standardized with a block size of 100 and 64 available threads. Furthermore, we included the optimal configuration of FISCO-BCOS (block size 1000) in our environment for comprehensive comparison. In comparison to ChainMaker, BachLedger demonstrates a significant performance advantage due to its utilization of the OEV architecture, which enables full parallelization of consensus and other procedural steps. In contrast, ChainMaker, which employs an EOV architecture, necessitates the completion of block execution prior to reaching consensus on blocks containing execution results, in order to ensure deterministic outcomes.

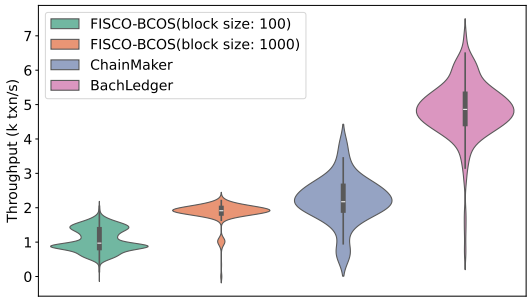

Fig. 6. End-to-end performance distribution comparison.

In the comparison with FISCO-BCOS, BachLedger achieves enhanced performance through the optimization of thread idle time. In extreme scenarios where a block contains transactions exclusively related to a single account, these transactions can only be executed sequentially due to logical constraints. Encountering such situations, FISCO-BCOS must wait for synchronization, thereby generating fragmented thread idle time, BachLedger allows subsequent blocks to utilize these idle resources efficiently. This optimization strategy enables BachLedger to maintain high performance even in challenging scenarios, further distinguishing it from existing high-performance blockchain systems.

C. Scalability Experience

This section evaluates the efficacy of our Seamless Scheduling design in enhancing system scalability. Our research reveals an optimal block size exists, as both excessive and insufficient sizes impair system performance. This optimum varies across blockchain implementations due to diverse algorithms and pipeline parallelization designs. We examine system efficiency, measured by throughput, as parallel computing resources increase. To ensure fair comparison across resource conditions, we introduce ρ, the ratio of available hardware cores to transactions per block, quantifying relative resource abundance. Figure 7 illustrates BachLedger's throughput as a function of ρ, demonstrating significant performance scaling with increased resource availability and evidencing BachLedger's efficient utilization of additional computational resources.

It is worth noting that BachLedger's Seamless Scheduling exploits fragmented idle resources arising from stochastic transaction execution times, resulting in probabilistic performance gains. While abundant resources increase the likelihood of achieving higher throughput, the performance lower bound remains constrained by inter-transaction dependencies.

VI. Conclusion

Our theoretical and empirical investigations reveal that, with the increasing parallel computing capacity of blockchain

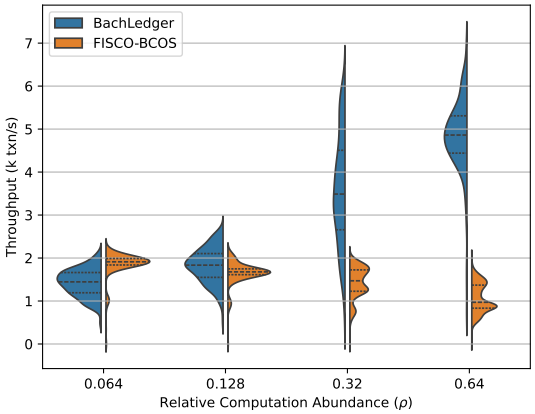

Fig. 7. Throughput distribution comparison at varying ρ.

nodes, the proportion of thread idle time resulting from intra-block synchronous execution is growing substantially, significantly impacting overall performance. This paper presents BachLedger, which introduces a novel Seamless Scheduling algorithm to efficiently harness these fragmented computational resources. Experimental results demonstrate that BachLedger's implementation of this algorithm yields substantial performance improvements over current state-of-the-art solutions.

References

[1] Y. Chen *et al.*, "Fangorn: Adaptive execution framework for heterogeneous workloads on shared clusters," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2972–2985, Jul. 2021.

[2] T. Li, S. Ying, Y. Zhao, and J. Shang, "Batch Jobs Load Balancing Scheduling in Cloud Computing Using Distributional Reinforcement Learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 1, pp. 169–185, Jan. 2024.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[4] X. Qi *et al.*, "SChain: Scalable Concurrency over Flexible Permissioned Blockchain," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, Apr. 2023, pp. 1901–1913.

[5] H. Li *et al.*, "FISCO-BCOS: An Enterprise-grade Permissioned Blockchain System with High-performance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, Nov. 2023, pp. 1–17.

[6] C. Wu, M. J. Amiri, J. Asch, H. Nagda, Q. Zhang, and B. T. Loo, "FlexChain: An elastic disaggregated blockchain," *Proceedings of the VLDB Endowment*, vol. 16, no. 1, pp. 23–36, Sep. 2022.

[7] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18, Apr. 2018, pp. 1–15.

[8] J. Qi *et al.*, "Bidl: A High-throughput, Low-latency Permissioned Blockchain Framework for Datacenter Networks," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, Oct. 2021, pp. 18–34.

[9] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang, "Fine-grained, secure and efficient data provenance on blockchain systems," *Proceedings of the VLDB Endowment*, vol. 12, no. 9, pp. 975–988, May 2019.

[10] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, May 2020, pp. 543–557.

[11] *Monad*, https://www.monad.xyz/, 2024.

[12] Z. Peng *et al.*, "NeuChain: A fast permissioned blockchain system with deterministic ordering," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2585–2598, Jul. 2022.

[13] R. Gelashvili *et al.*, "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*, Feb. 2023, pp. 232–244.

[14] *Chainmaker*, https://chainmaker.org.cn/, 2021.

[15] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "Dispersedledger:high-throughput byzantine consensus on variable bandwidth networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 493–512.

[16] Q. Wang and R. Li, "A weak consensus algorithm and its application to high-performance blockchain," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, IEEE, 2021, pp. 1–10.

[17] *Slither*, https://github.com/crytic/slither, 2021.

[18] C. Xu, C. Zhang, J. Xu, and J. Pei, "Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2314–2326, 2021.

[19] J. Chen and S. Micali, "Algorand," *arXiv preprint arXiv:1607.01341*, 2016.

[20] N. Ivanov, Q. Yan, and Q. Wang, "Blockumulus: A scalable framework for smart contracts on the cloud," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2021, pp. 607–617.