Pythia: Supercharging Parallel Smart Contract Execution to Guide Stragglers and Full Nodes to Safety

Ray Neiheiser
ISTA, Austria

Arman Babaei
ISTA, Austria

Ioannis Alexopoulos
ISTA, Austria

Marios Kogias
Imperial College London, England

Eleftherios Kokoris Kogias
Mystenlabs

Abstract

Blockchain performance has historically faced challenges posed by the throughput limitations of consensus algorithms. However, recent breakthroughs in research have successfully alleviated these constraints. One of the key elements is the introduction of a modular architecture that decouples consensus from execution. Due to these recent advances, attention has shifted to the execution layer.

While parallel transaction execution is a promising solution for increasing throughput, practical challenges persist. Its effectiveness varies based on the workloads, and the associated increased hardware requirements raise concerns about undesirable centralization, given that already over 30% of Ethereum nodes are unable to keep up. These increased requirements result in full nodes and stragglers synchronizing from signed checkpoints, decreasing the trustless nature of blockchains.

In response to these challenges, this paper introduces Pythia, a system designed to extract execution hints for the acceleration of straggling and full nodes. Notably, Pythia achieves this without compromising the security of the system or introducing overhead on the critical path of consensus. Evaluation results demonstrate a notable speedup of up to 60%, effectively addressing the gap between theoretical research and practical deployment. The quantification of this speedup is achieved through realistic blockchain benchmarks derived from a comprehensive analysis of Ethereum and Solana workloads, constituting an independent contribution.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases Blockchain, Parallel Execution, Smart Contracts, Benchmark, Stragglers

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Due to recent research efforts reaching visa-level throughput for Byzantine Fault-Tolerant Consensus [7, 13, 23, 1, 17], the performance of smart contract execution has shifted into focus. This is particularly relevant as many blockchains, such as Ethereum [4], still execute transactions sequentially, not taking advantage of modern multicore architectures.

Realizing this new challenge, parallel execution engines have emerged [11, 25] that allow for parallel rather than sequential processing. In many practical deployments of these new execution engines, such as in Solana, Aptos, or Sui [27, 10, 26], execution is decoupled from consensus, removing the execution from the critical path of consensus. Although this results

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in a dirty ledger [24] where transactions, even though included in a block, might still be aborted during execution, this approach has shown to improve throughput significantly [7].

These parallel execution engines can be divided into two main categories: *Optimistic* and *Guided* execution engines [11, 27, 21, 8]. Guided execution engines, as used in Solana [27] and Sui [26], rely on an exhaustive set of resource addresses, often referred to as hints, that the client has to send alongside the transaction. The execution engine then schedules the transactions for execution while accounting for potential read-write conflicts, guaranteeing at the application level that no conflicts may arise. If a transaction fails to exhaustively declare its dependencies, the execution engine detects the out-of-bounds access and aborts the transaction. In the database context, this is comparable to pessimistic approaches, where locking is used to prevent conflicts.

Meanwhile, optimistic execution engines, as used in Aptos [10, 11], optimistically execute transactions in parallel, detect conflicts as they arise, and re-execute transactions when necessary. While, in the worst case, optimistic execution engines may have a high re-execution overhead, they simplify application development, as applications don't have to provide execution hints and they can be integrated more easily into existing blockchains where the concept of execution hints does not exist [25]. However, the actual overhead of fully optimistic execution is still unclear, as the Polygon team observed a minimal speedup compared to sequential execution due to the nature of their blockchain workload [25].

Inspired by this, as a first contribution in this work, we identify and close a clear gap between theoretical research and practical deployments; the lack of realistic blockchain workloads that correctly capture the type and frequency of data dependencies and contention. This is essential as the high levels of contention we identified in our analysis, significantly affect the effectiveness of the execution engine.

Additionally to the unclear practical impact, parallel transaction execution comes with a second significant caveat: it is a force for centralization. Modern blockchains utilizing concurrent execution also have higher hardware requirements, and to make matters worse run so fast that struggling consensus nodes and full nodes have no choice but to catch up, not by auditing and re-executing the transactions, but by simply downloading signed checkpoints. This is especially staggering given that, at the time of writing, over one-quarter of all Ethereum nodes are unable to keep up for diverse reasons at lower hardware requirements and throughput [6].

Catching up is important for several reasons. First, struggling full nodes will only respond with significant delays to their clients. Struggling validators might not be able to propose or propose a significant number of stale transactions, reducing the throughput of the system. Furthermore, in the presence of faulty nodes, the system might stall until all correct validators catch up to the head of the chain.

When catching up through signed checkpoints only a smaller subset of active nodes verify the validity of the blocks in the chain. While this is consistent with the standard BFT assumptions, where for any number of faulty nodes $f' > f$ the system cannot guarantee safety, in practical deployments there are recovery mechanisms that allow even a minority of correct validators to eventually re-establish safety after a successful attack [2, 3]. However, this requires correct and active validators to verify all blocks in the chain to identify validity violations or equivocations and initiate the recovery protocol. Furthermore, these validity guarantees only extend to the client if full nodes also execute and verify all blocks before providing them to their clients. This is also consistent with the initial Bitcoin vision for full nodes and validators [16, 20, 18]. We denote this property as *unconditional validity.*

In this paper, we propose Pythia, a framework that helps struggling validators and full nodes, hereafter denoted as stragglers, catch up in blockchains that deploy optimistic execution engines as, for example, Aptos [10], Monad [15] or Sei [22], without sacrificing unconditional validity. Pythia achieve this by bridging the gap between optimistic and guided execution engines by extracting accurate hints during the actual execution and providing them to stragglers to catch up.

Pythia does not rely on hints for safety and does not introduce any overhead on the critical path of consensus. This reestablishes the initial security vision of Bitcoin with minimal overhead. Our evaluation of Pythia shows that stragglers can, depending on the workload, execute blocks up to 60% faster.

In summary, we provide the following contributions:

- We propose Pythia, a framework to enable stragglers to catch up through guided execution without relaxing security guarantees.
- We construct a microbenchmark for parallel smart contract execution engines based on real-world data.
- We evaluate Pythia under the proposed microbenchmark and show a speed up of up to 60% compared to optimistic parallel execution.

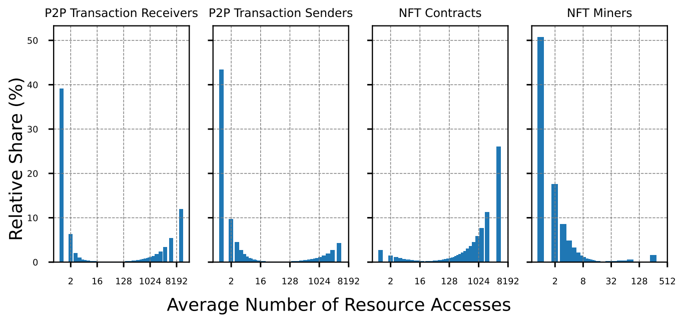
2 Blockchain Workloads

Most approaches in academia and industry that evaluate parallel smart contract execution engines either generate random, artificial peer-to-peer transfers [11] with uniform distributions or apply non-blockchain-based workloads [12] such as Uber's, Youtube's, or Twitter's. However, these do not reflect the real-world workloads that production blockchains are subject to. Due to this, these benchmarks are unable to highlight the shortcomings of existing parallel smart contract execution engines. While some works [25] evaluate the performance of the execution engine by re-executing a part of the past transaction history, this approach has two important limitations. First, the blockchains these workloads stem from do not natively support parallel execution, and, therefore, the smart contracts were not developed with concurrency in mind. Second, these workloads are restricted to their respective ecosystems and cannot easily be ported to a different blockchain to compare the performance of two competing approaches from different ecosystems. In contrast, for our workloads, we extract the essential points of contention from different application settings to guarantee easy portability to diverse frameworks and virtual machines.

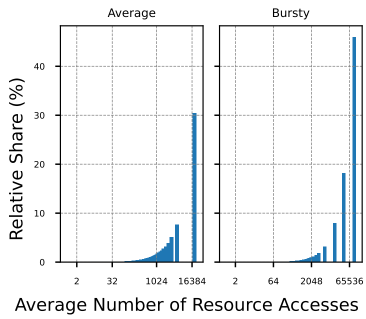
In this section, we analyze the transaction history of popular blockchains and blockchain applications. The purpose of this analysis is twofold: First, to identify acceleration opportunities for Pythia through careful execution scheduling based on hints, compared to existing optimistic parallel execution engines. Second, to provide a set of realistic and versatile blockchain workloads for our evaluation and make them accessible to the community.

Therefore, we conducted a thorough analysis of the user activity on Ethereum, the most well-known smart contract ecosystem, and Solana, the most prominent blockchain that supports parallel execution. As a result, we identified four realistic and popular blockchain execution scenarios: NFT Minting, DEX Trading, Peer-to-Peer (P2P) Transactions, and a Mixed Workload.

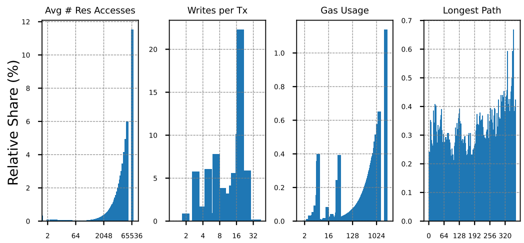
These scenarios cover a wide range of execution characteristics, from heavy contention and complex contract interactions to simple P2P transactions. This allows for a more comprehensive evaluation of execution engines and their ability to handle the demands in a real-world blockchain setting.



(a) Ethereum Workloads



(b) DEX Workloads



(c) Mixed Workload

To create the benchmarks, we analyzed the average distribution of resource accesses throughout 2022. Since transactions are not time-independent, instead of calculating the yearly average, we computed the average access distribution across every 1000 blocks for the NFT, P2P, and Mixed workloads. For the DEX workload, we performed a day-by-day assessment. To account for variability, we split it into two workloads: one representing the average daily volume and another representing the 30 most contended days of the year.

In the following, we describe the workloads in detail:

2.1 Peer-to-Peer Transaction Workload

First, the *Peer-to-Peer Transaction* workload. Instead of assuming a uniform distribution, we measured the distribution of senders and receivers of payment transactions on Ethereum. The result is displayed in Figure 1a. On the x-axis, we display resource frequency groups (e.g. $1, 2, 10, ..$ resource accesses per 1000 blocks) and on the y-axis the percentage share of each of the groups. Even though the largest group is independent resource accesses (around 40% of both receivers and senders only appear on average once every 1000 blocks), over 10% of the transactions involve the same account.

Algorithm 1 *P2P Smart Contract*

```
resourcetable ← ∅
procedure ACCESSTWO(addr1, addr2)
    for i = 1 → R do
        resourcetable[addr1] += 1
        resourcetable[addr2] += 1
    end for
end procedure
```

P2P transactions always at least conflict on the sender balance and on the receiver balance. We, therefore, created a very simple smart contract for our benchmark, that can be ported trivially to any smart contract language. Algorithm 1 shows the pseudocode. The smart contract holds a resource table, and each transaction accesses two resources (i.e. sender and receiver balance). We do the table assignment in a loop of R iterations to simulate a realistic runtime, given that in most blockchain ecosystems there is a comprehensive number of checks (reads and writes) for each P2P transaction. This parameter can be obtained by comparing the execution time of the simplified smart contract with the runtime of a regular transaction in the given ecosystem.

2.2 NFT Workload

Next, the *NFT Minting* workload is derived from Ethereum's minting behavior in 2022. The distribution of smart contract accesses and miner addresses is also shown in Figure 1a. There is already significantly more contention in this workload, as the two most popular NFT smart contracts combined show up in over 35% of all transactions while only a small number of smart contracts are only accessed once within the same period. Meanwhile, the accounts minting the NFTs are well distributed, and over 50% of users only minted one NFT within the 1000 block period.

Algorithm 2 *Single Resource Smart Contract*

```
resourcetable ← ∅
procedure ACCESSONE(addr1)
    for i = 1 → R do
        resourcetable[addr1]+ = 1
    end for
end procedure
```

In the case of NFT minting, we expect each transaction that mints the same NFT to conflict due to the NFT index that is incremented with each transaction. This index is also usually used to limit the number of NFTs. The pseudocode is displayed in Algorithm 2. Each transaction accesses a single resource in the table and increments the value R times, analog to the P2P workload to simulate a realistic runtime complexity.

2.3 Decentralized Exchange Workload

In the context of decentralized exchanges, we created two *DEX Workloads* for which we gathered data on the daily distribution of different trading pairs on Uniswap [14] throughout 2022. The first workload is an *Average DEX Workload* derived from the annual average. As we observed a large variance in the daily distribution of trading behavior, we created a second workload, termed a *Bursty DEX Workload* that we computed based on the average over the thirty most contended days. The results of this analysis are depicted in Figure 1b where the x-axis represents the frequency of unique pairs and the y-axis the percentage share of this group. We observe that on average, over 30% of all transactions trade the same coin pair, while on the 30 most contended days, over 45% of the transactions trade the same coin pair, and the three most popular trading pairs make up over 70% of all transactions.

In DEX smart contracts, each transaction for a given coin-pair at least touches the same liquidity pool, as such, at least transactions trading the same coin-pair on a given dex necessarily have to conflict. Therefore, we use the same smart contract as in Algorithm 2.

2.4 Mixed Workload

Finally, for the *Mixed Workload*, we extracted the write sets of Solana transactions and their corresponding gas expenditures. This workload is the most complex among the four, as it involves varying the length of the write-set, the access distribution of resources within the write-set, and the transaction runtime. Due to the large number of blocks that are produced every day on Solana, we only queried a representative sample of 1000 blocks per day throughout 2022 and discarded the system maintenance transactions. The results are shown in Figure 1c. Analogous to the other workloads, we observe that a small number of resources make up a large percentage of the write accesses. Furthermore, we observe that most transactions access several resources and that the execution times are widely distributed. The access distribution of the resources results in critical paths taking up between 20 and 60% of all transactions with an average of around 30%. This was calculated by comparing the total gas consumption for 1000 blocks with the combined gas cost of the longest path of dependent transactions within the same period. To make sure that the critical path in the benchmark approximates what we observed in the data, we adjusted the workload generation code such that the resource distribution, number of writes, and transaction length, on average result in a critical path of around 30%.

Algorithm 3 *Multi Resource Complex Runtime Smart Contract*

```
resourcetable ← ∅
procedure ACCESSN(complexity, setofaddresses)
    i ← 0
    for all addr ∈ setofaddresses do
        i+ = 1
        resourcetable[addr]+ = 1
    end for
    for j = i → complexity do
        setofaddresses[j%|setofaddresses|]+ = 1
    end for
end procedure
```

The mixed workload actively accesses a range of resources a varying number of times to vary the smart contract runtime. The pseudocode for the smart contract is outlined in Algorithm 3. In a loop, we access the resource table and increment the value of each resource in the set of addresses at least once. Next, following the complexity parameter, we iterate an additional $complexity - i$ times and access the addresses in the set of addresses uniformly.

2.5 Summary

With the help of the access distribution, we sample transactions from the respective workloads to generate the microbenchmarks. For example, if 10% of transactions in the workload access the same resource and the remaining 90% access independent resources, the microbenchmark reflects this distribution. As such, given the 10% example, the probability of having two transactions accessing the same resource back-to-back in the block is 1%. Note that, given our workload analysis, which shows that independent users regularly access the same resources, we believe this correctly reflects the average distribution even from a more fine-grained perspective.

The benchmarking code and instructions are available at `https://anonymous.4open.science/r/execution-engine-benchmark-C229`. For each workload, we provide datasets to construct probability distributions, represented as sets in the form $[1, 1, 1, 1, 10, 100]$, where each entry corresponds to a specific resource, and the value indicates the probability of that resource being accessed. To generate the workload, resources are selected iteratively based on their weighted probabilities within the distribution.

Summarizing, our workload analysis shows that except for the peer-to-peer workload, all workloads are highly contended, validating our initial claim. Due to these dependencies, naively executing smart contracts in parallel will lead to high abort rates. PYTHIA leverages this and uses dependency hints to guide the parallel execution at stragglers and full nodes.

In the remainder of this paper, we discuss the design of PYTHIA that allows maximizing parallelism in this context.

3 System Model

Before delving into the design of PYTHIA, let's first explore the underlying system model upon which we construct PYTHIA. We assume the existence of a set of N server processes $p_1, p_2, ..., p_N$ and a set of I client processes $c_1, c_2, ..., c_I$ communicating over a peer-to-peer network where clients and servers are identified with the help of asymmetric key pairs [16] and entities prove their identity by signing their respective transactions and messages. Furthermore, clients and servers communicate over perfect point-to-point channels achieved through mechanisms for message retransmissions, ordering, and deduplication. Thus, if a process p_i sends a message m_{ij} to process p_j, p_j eventually receives m_{ij}.

To deal with network failures, we assume that the network follows a partial synchrony model based on [9]. While, during periods of asynchrony, messages may be delayed for an arbitrary amount of time, we assume the existence of regular periods of stability, after some *Global Stabilization Time* (GST). During these periods, messages passed between two correct processes arrive within a known bound δ.

Similar to the current state-of-the-art [7, 10], execution and consensus are split into modular layers that run in parallel such that execution does not happen on the critical path of consensus. Within the consensus layer, we treat consensus as a Black-Box, where each node receives an identical chain of blocks $B_1, B_2, ..., B_i$ which is then executed sequentially. As long as the execution is strictly deterministic, this approach ensures that all server processes reach the same state. However, as execution does not occur on the critical path of consensus, client transactions cannot be fully validated before block inclusion. Thus, we assume the output of consensus to be a *dirty-ledger* [24] where the resulting blockchain might contain invalid transactions (e.g. lacking funds or gas). Nonetheless, as long as the executors receive the same blocks in the same order (guaranteed by consensus) and the output of the execution of the chain of blocks is strictly deterministic (i.e. all executors invalidate the same transactions), all executors produce the same output.

To ensure determinism in the context of parallel execution, which is necessary to guarantee safety, we adopt a structure similar to BlockSTM [11], where transactions go through two distinct phases. In the execution phase, transactions are executed optimistically, and in the validation phase, the execution read and write sets are cross-validated with the help of a multi-version data structure. Validation and execution operate concurrently and if inconsistencies are detected transactions are rolled back and rescheduled for execution.

Finally, we assume a Byzantine fault model inherited from the chosen consensus framework, with the number of faulty nodes bound by $N = 3f + 1$ as in most permissioned consensus algorithms. A given process is considered correct as long as it follows the protocol, otherwise, it is deemed faulty.

4 Design Overview

Drawing from our workload analysis and the insight into the impact of block composition on performance, we establish specific design objectives for PYTHIA. First, we want to speed up execution in a way that maximizes parallel execution and avoids frequent re-executions without relaxing safety. Second, we want stragglers and full nodes to be able to query and verify hints with minimal overhead.

In section 2 we show that the workloads we expect in a blockchain setting are highly contended, hence leveraging hints from the execution result can eliminate re-executions. In this section, we outline how hints are extracted and propagated and how nodes catch up with the help of the hints without sacrificing safety.

4.1 Hint Extraction

The optimistic parallel execution engine BlockSTM [11] tracks all transactions with the help of a multi-version data structure that records the read- and write-sets of all transactions. This allows BlockSTM to detect conflicts between concurrently executed transactions during run-time and enables it to initiate re-execution with fresher inputs when necessary.

After deterministically finishing the execution of all transactions, the read and write-sets are extracted from the multi-version data structure as the resulting state from the parallel execution, alongside a footprint measured in gas that represents the execution complexity.

This data is then stored alongside the block such that it can be sent to straggling nodes and full nodes on request. To preserve bandwidth and storage, this data can be compressed into solely tracking the closest dependency of a given transaction. I.e. if transaction B depends on A and transaction C depends on A and B, transaction C only has to declare its dependency on B, as B already declares its dependency on A.

4.2 Hint Propagation and Catching Up

In order to receive execution hints, full nodes or straggling nodes contact active nodes and request hints. Depending on how far behind a node is, it can obtain hints for several blocks or, if it is close to catching up, for a single block at a time. There are two ways how nodes can detect they are straggling. Either, by evaluating the progress of the other nodes in the network from the incoming execution state commit certificates as used in Aptos, or by evaluating the number of blocks the node has queued for execution. In either case, nodes can then request execution hints to speed up their execution to catch up with the remaining nodes in the system.

Invalid hints might omit transaction dependencies leading to transaction re-execution, or include unnecessary dependencies leading to a sequential execution and harming performance. While Pythia still uses the validation step of BlockSTM to prevent this from impacting safety, incorrect hints could result in straggling nodes falling further behind, impacting the system latency and throughput significantly. However, as we prove in the following, as long as execution hints are signed by at least $f + 1$ nodes, they are guaranteed to be correct.

▶ **Theorem 1.** *Execution hints signed by $f + 1$ nodes are guaranteed to be correct given $N = 3f + 1$.*

Proof. The proof is straightforward. Given $f + 1$ execution hints for a given block, at least 1 of the hints must've been provided by a correct node. Given the safety requirements of byzantine fault tolerant consensus [5], no two correct nodes will decide on conflicting values. As such, as long as the used consensus framework is safe, and correct nodes only execute blocks that were output by consensus, any set of execution hints signed by at least one honest node must be correct. ◀

Nonetheless, as discussed in the introduction, based on the trust model of Bitcoin, we want the system to be able to recover from more severe failures where potentially more than f nodes are faulty and might provide invalid hints to straggling nodes. This is still

Algorithm 4 *Transaction Scheduling*

```
 1: queue ← ∅
 2: priorityqueue ← ∅
 3: depgraph ← ∅                    ▷ Parent/Child Relationship between transactions
 4: procedure SCHEDULE(txns)
 5:     for all tx ∈ txn do                                ▷ Iterate over transactions
 6:         if |depgraph_tx.parents| ≤ 0 then
 7:             if |depgraph_tx.children| ≤ 0 then
 8:                 queue ← tx
 9:             else
10:                 priorityqueue ← tx
11:             end if
12:         else
13:             critparent ← ⊥
14:             for all tx_p ∈ depgraph_tx.parents do       ▷ Iterate over parent transactions
15:                 if critparent = ⊥ ∨ tx_p.pathcost > critparent.pathcost then
16:                     critparent ← tx_p                      ▷ Add as critical Parent
17:                 end if
18:             end for
19:             depgraph_critparent.primarychildren ← tx
20:         end if
21:     end for
22: end procedure
23: procedure EXECUTE(tx)
24:     for all tx_p ∈ depgraph_tx.parents do               ▷ Iterate over parent transactions
25:         if tx_p.status ≠ Completed then
26:             depgraph_tx_p.primarychildren ← tx             ▷ Add as critical Parent
27:             RETURN                                          ▷ Don't execute
28:         end if
29:     end for
30:     tx.execute                                            ▷ Execute Transaction
31:     for all tx_c ∈ depgraph_tx.primarychildren do         ▷ Iterate over critical children
32:         if |depgraph_tx_c.children| ≤ 0 then
33:             queue ← tx
34:         else
35:             priorityqueue ← tx
36:         end if
37:     end for
38: end procedure
```

possible due to the validation step in the catch-up mode where nodes can detect invalid hints, broadcast a proof of misbehavior, and update their trust assumptions accordingly (e.g. ignore future execution hints from the $f+1$ nodes that previously signed invalid hints). Due to this, as long as there is at least one honest non-straggling node every straggling node will eventually be able to connect to it and catch up using its hints.

As hints are a result of the execution, stragglers and full nodes will always be behind the head of the chain by at least some $\Delta = \alpha + \delta$ where α denounces the execution delay and δ the transmission delay. However, as hints can be obtained from a subset of correct nodes, δ can be kept minimal when hints are requested from geographically close nodes.

4.3 Guided Parallel Execution

Based on the dependency graph and the execution time of each transaction an execution schedule that avoids re-executions and prioritizes long chains of transactions can be built. The algorithm on how this is done is presented in Algorithm 4.

In a nutshell, all the dependencies of transactions are checked and if a transaction has no parents, it can be scheduled for execution. If it has child transactions it gets into the priority pool, to prioritize executing long chains first, otherwise, the transaction gets added to the regular queue (Line 7).

Given the path cost of each transaction, i.e. the sum of the cost of the longest path leading to a given transaction including the transaction's own cost, the *critical parent* can be computed (i.e. the parent that will take the longest to finish executing). All children of the *critical parent* are added as *primary child* to their parent (Line 19), and after the *critical parent* finishes execution, all child transactions can be scheduled for execution in the respective queues (Line 32).

However, as this is done optimistically before any transaction is executed it is necessary to verify that all parent transactions already finished executing; if a parent that has not finished executing yet is found, the transaction is added as a child of this parent, and its execution is delayed (Line 26).

For safety reasons, concurrent to execution, the validation procedure of BlockSTM [11] is followed. For each executed transaction, its read set is compared to the write sets of the preceding transactions to scan for potential conflicts or invalid hints. In case a conflict arises, the transaction is rescheduled for execution and the system falls back to BlockSTM for the remaining transactions and, the nodes that provided the hints are marked as untrustworthy. As validation reads from the multi-version data structure and does not require spawning a virtual machine, it takes a fraction of the time of the execution.

As guided execution prevents frequent re-executions, in the presence of long critical paths of transactions, we expect many CPU cores to remain idle during parts of the execution. This opens a window for other optimizations such as leveraging the idle cores to verify transaction signatures instead of doing so on the critical path of execution. We outline these optimizations in Section 5.1.

Finally, by re-using the same validation phase of BlockSTM, and, as such, only altering the execution order, Pythia also inherits the determinism guarantees of BlockSTM, and as such guarantees safety.

5 Evaluation

5.1 Implementation

We implemented Pythia on top of BlockSTM [11] in Aptos [10] with two major alterations. First, we replaced the BlockSTM scheduler with our guided execution scheduler from Section 4.3 and second, we implemented the hint extraction from the execution results.

Alongside this, we also added several optimizations that opened up due to the new scheduling approach. Traditionally, client transaction signatures are verified before handing the transaction to the execution engine. In Pythia, as the scheduler is aware of the dependency chains, we can leverage idle execution workers to verify client transaction signatures during execution, moving the signature verification away from the critical path. While BlockSTM is unaware of the transaction dependency graph, there are some sequential parts in the BlockSTM scheduler where idle workers could also be leveraged to verify client transactions. Therefore, to allow for a fair comparison we also implemented the signature verification optimization for BlockSTM.

Furthermore, in Aptos, transaction execution is split into three phases: Prologue, the actual execution, and epilogue. For each part a virtual machine has to be instantiated, and, as such, for small transactions, each takes up roughly a third of the total execution time. As conflicts here only occur when the transaction is from the same user, BlockSTM almost fully parallelizes the prologue and epilogue, reducing the potential speedup Pythia can achieve in practice.

To compensate for this, we added additional functionality to Pythia, to execute transaction prologues in parallel if they are the only transaction of a given client in the block, the first transaction of a given client in the block, or, the previous transaction of the client has been completed (prologue, execution, and epilogue).

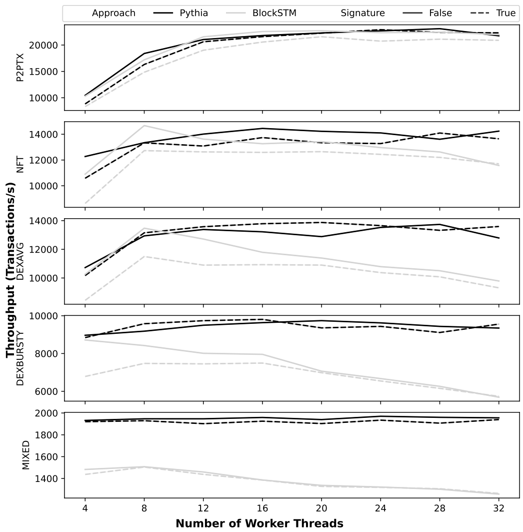


Figure 2 Throughput per Second - Execution Engine

Next, to take advantage of cache locality, when a worker thread schedules the next set of child transactions, it selects one destined for the priority queue and places it at the top of its local queue.

Finally, we implemented the workloads mentioned in Section 2 in Move Contracts and added code to benchmark the execution engine with and without hints under the proposed workloads.

5.2 Benchmark

We executed our experiments on a Debian GNU/Linux 12 server with two AMD EPYC 7763 64-Core Processors and 1024 GB of RAM. We created blocks of 10000 transactions with the help of our benchmarking suite and subsequently submitted them to the BlockSTM and Pythia executor. For each benchmark we created 16 different configurations; With and without the idle-worker signature verification for 4,8,12,16,20,24,28 and 32 worker threads. We executed each configuration a total of 10 times and then computed the average.

Figure 2 shows the per-second throughput for all workloads for the baseline BlockSTM in gray and the guided parallel execution Pythia in black without signature validation (full line) and with idle-worker signature validation (dotted line). The y-axis depicts the throughput in transactions per second and the x-axis depicts the different configurations of worker threads from 4 to 32.

We ordered the workloads by the level of contention, starting from the top with the P2P workload. Unsurprisingly, for both the P2P and NFT workloads, at lower levels of contention and small transactions, the speed-up of Pythia is minimal. However, Pythia still shows a small performance advantage for all configurations with the idle-worker signature verification.

However, with increasing contention, the performance advantage becomes increasingly clear, especially when including the idle-worker signature verification. For all three remaining workloads, Pythia already shows a significant performance advantage at 4 worker threads which increases further with the number of worker threads. This is the case, as with high levels of contention BlockSTM spends an increasing amount of time re-executing and revalidating transactions in a cascading fashion, which results in a performance drop with an increasing number of worker threads. As Pythia is aware of the dependency graph, it is able to stabilize the performance and use the idle time to verify transaction signatures and pre-execute the transaction prologue.

While for the DEX workloads, most of the advantage at the lower number of worker threads comes from the idle-worker signature verification, when the execution is more computationally intensive like in the MIXED workload, the influence of the idle-worker signature verification is much smaller. This is the case as the accidental parallelization of the Prologue and Epilogue in BlockSTM plays a much smaller role, allowing Pythia to outperform BlockSTM even without this optimization significantly by 40% at 4 cores up to 60% at 32 cores.

As such, in any configuration, by using Pythia, straggling nodes are able to leverage hints to catch up to the remaining nodes. Furthermore, the more computationally intensive and contended the workloads are the more the speed-up performance increases. This is especially important as we expect nodes to be more likely to struggle to keep up when the workloads are more computationally intensive.

6 Related Work

We divide related work into two categories. Benchmarks for smart contract execution engines, and approaches related to catching up in the context of guided parallel execution.

6.1 Execution Engine Benchmarks

One of the most comprehensive benchmarks for blockchains is Diablo [12] which offers a full benchmark suite for blockchain evaluation. However, we identify several shortcomings. First, the workloads Diablo offers do not correspond to typical blockchain workloads (e.g., large data upload tasks or computationally intensive tasks). Furthermore, the chosen workloads also neither correspond to typical usage patterns in terms of user distribution nor regarding contention levels. This makes these workloads unsuitable for evaluating the practicability of parallel smart contract execution engines.

While newer approaches such as [19] offer benchmarks based on real-world blockchain workloads, they lack a focus on evaluating parallel execution engines with realistic levels of contention (i.e., they only evaluate traditional blockchains with sequential execution).

As such, in contrast to previous work, we provide a set of workloads that mirror real-world contention levels with the goal of evaluating the effectiveness of parallel smart contract execution engines. As most blockchains still use sequential execution, smart contracts are usually not written with concurrency in mind, introducing artificial contention that can be fixed on the application level. Therefore, we've created a set of smart contracts that only generate conflicts on the storage elements that can not be easily parallelized.

Furthermore, instead of building a full benchmarking suite, we opted for providing a set of simple probability distributions that can be used alongside a set of simplified smart contracts to allow easy incorporation of our workloads in any benchmarking suite. This facilitates using our workloads to compare different types of smart contract platforms.

6.2 Catching up under Parallel Execution

Existing blockchains that implement parallel transaction execution, such as Aptos [10] or Sui [26] only allow stragglers to catch up with the help of signed execution results and checkpoints that were signed by a significant percentage of the stake (i.e. typically $f + 1$ or $2f + 1$ out of $N = 3f + 1$). However, as already pointed out, this presents a relaxation of the security guarantees compared to Bitcoin from at least one honest to at least $f + 1$ honest. This is the case as straggling validators and full nodes become unable to verify the correctness of the execution state, significantly reducing the number of nodes that can detect misconduct in the presence of large numbers of faulty nodes.

Hint-driven execution is a common approach in the parallel execution space. Solana and Sui [27, 26] leverage client hints to build a parallel execution schedule preventing frequent re-executions at the cost of higher application development complexity and overly pessimistic hints reducing the parallelizability of the workload.

Meanwhile, Polygon [25] and Dickerson et al. [8] have miners pre-execute the block on the critical path of consensus and leverage hints resulting from this process to speed up the actual execution. However, this comes at a large overhead as regardless of how fast the actual execution is, the optimistic execution on the critical path of consensus will slow down the system significantly. This makes matters worse for straggling validators as the large pre-execution overhead in combination with lagging behind makes it unfeasible to propose a block in the expected time frame. As a result, this further slows down the entire system.

Therefore, to the best of our knowledge, Pythia is the first framework that allows straggling nodes to catch up in a modular parallel smart contract execution environment without relaxing the security guarantees and without introducing additional complexities for application developers. In addition to that, compared to existing hint-based solutions Pythia provides several novel strategies to maximize the throughput given the execution hints. First, to avoid long critical paths clogging up the systems, transactions with a chain of dependent transactions are executed with a higher priority. Furthermore, we leverage idle CPU cores during execution, resulting from the optimized schedule to verify client transaction signatures instead of verifying them on the critical path of execution.

7 Conclusion

In this work, we presented Pythia, a guided parallel execution engine that allows straggling nodes and full nodes to catch up without relaxing the security guarantees. Depending on the workload Pythia outperforms the optimistic execution engine by up to 60%.

Furthermore, we created a set of realistic workloads based on real-world data to evaluate the effectiveness of parallel execution engines to allow comparing parallel execution engines of diverse eco-systems even if they use different virtual machines and programming languages.

Furthermore, Pythia opens the doors for a range of future work such as evaluating its effectiveness in the context of parallel execution engines such as Solana [27] or Sui [26] where client hints might be overly pessimistic. Finally, our workload analysis has shown that the execution layer is still one of the major bottlenecks and hurdles to scaling blockchain systems, warranting further studies and novel approaches to further alleviate this bottleneck.

References

1 Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path, 2023. `arXiv:2310.14821`.

2 Christian Badertscher, Yun Lu, and Vassilis Zikas. A rational protocol treatment of 51% attacks. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, pages 3–32, Cham, 2021. Springer International Publishing.

3 Eric Budish, Andrew Lewis-Pye, and Tim Roughgarden. The economic limits of permissionless consensus, 2024. URL: https://arxiv.org/abs/2405.09173, arXiv:2405.09173.

4 Vitalik Buterin. Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 2013. Accessed on 12.04.2023.

5 C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, Berlin, Heidelberg, 2nd edition, 2011.

6 ChainSafe Systems Inc. General information. https://nodewatch.io, 2024. Accessed on 08.01.2024.

7 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519594.

8 Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 303–312, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3087801.3087835.

9 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 4 1988.

10 Aptos Foundation. Aptos whitepaper. https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf, 2023. Accessed on 12.04.2023.

11 Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022. URL: https://arxiv.org/abs/2203.06871, doi:10.48550/ARXIV.2203.06871.

12 Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 540–556, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3552326.3567482.

13 Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 803–818, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417262.

14 Uniswap Labs. Uniswap protocol. https://uniswap.org, 2024. Accessed on 08.01.2024.

15 Monad. Extreme parallelized performance. https://www.monad.xyz, 2024. Accessed on 24.07.2024.

16 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

17 Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 35–48, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483584.

18 Bitcoin Org. Full nodes. https://bitcoin.org/en/full-node, 2023. Accessed on 27.12.2023.

19 Kunpeng Ren, Jefferson F.B. Van Buskirk, Zheng Yong Ang, Shizheng Hou, Nathaniel R. Cable, Miguel Monares, Hank F. Korth, and Dumitrel Loghin. Bbsf: Blockchain benchmarking standardized framework. In *Proceedings of the 1st Workshop on Verifiable Database Systems*, VDBS '23, page 10–18, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3595647.3595649.

20 Ashish Rajendra Sai, Jim Buckley, Brian Fitzgerald, and Andrew Le Gear. Taxonomy of centralization in public blockchain systems: A systematic literature review. *Inf. Process. Manage.*, 58(4), jul 2021. doi:10.1016/j.ipm.2021.102584.

21 Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts, 2019. URL: https://arxiv.org/abs/1901.01376, doi:10.48550/ARXIV.1901.01376.

22 Sei. Accelerating the future. `https://www.sei.io`, 2024. Accessed on 24.07.2024.

23 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery. `doi:10.1145/3548606.3559361`.

24 Ertem Nusret Tas, Dionysis Zindros, Lei Yang, and David Tse. Light clients for lazy blockchains, 2022. URL: `https://arxiv.org/abs/2203.15968`, `doi:10.48550/ARXIV.2203.15968`.

25 Polygon Team. Innovating the main chain: a polygon pos study in parallelization. `https://polygon.technology/blog/innovating-the-main-chain-a-polygon-pos-study-in-parallelization`, 2022. Accessed on 05.12.2022.

26 The MystenLabs Team. The sui smart contracts platform. `https://docs.sui.io/paper/sui.pdf`, 2023. Accessed on 15.01.2024.

27 Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.