Written By Benjamin Funk

In the nine years since Ethereum launched the first decentralized, programmable blockchain, crypto has faced multiple roadblocks in the quest to scale decentralized applications to billions of users. And in order to develop scaling solutions to address this, the crypto industry has continuously funded and developed entirely new types of blockchains to solve the “performance problem.”

However, the “performance problem” has been poorly defined and quantified. Synthetic memes such as “transactions per second” have neatly packaged what are really apples-to-oranges comparisons between transactions that do not require equivalent computational work. The lack of nuance in these metrics also shrouds our ability to assess the independent impacts of a blockchain’s components on performance, distracting us from a principled approach to identifying the sets of optimizations we can make to solving highly interdependent problems.

Despite this fog, we have seen credible, sustained improvements to blockchain scalability play out over the past few years. As Ethereum powers through its rollup-centric roadmap, a new wave of rollups, coprocessors, data availability (DA) layers, and competing L1s are emerging – each with unique design choices to provide developers with more performant environments for building scalable, user-friendly dapps.

Today, the introduction of EIP4844 and alternative DA layers have alleviated the critical DA bottleneck. Despite this critical milestone, evidence suggests other important bottlenecks must be solved. Last month, Base collected $1.57M in transaction fees in a single daywhile paying only $5K in data availability costs to Ethereum. This suggests that the computational work required to validate and process state updates remains a critical bottleneck and an opportunity for improvement.

This piece will evaluate the design choices made by both integrated and modular execution environments in their path to solving for higher performance and expanding the scope of applications that can live onchain.

Today’s Challenges

The performance of an execution layer can be benchmarked according to the computational work that executing nodes achieve relative to their chains block time, or "gas computed per second.”

With this in mind, we can narrow execution layer bottlenecks to two interconnected factors: inefficient state access and inefficient computation.

Inefficient state access refers to the overhead of retrieving and updating the blockchain's state, which can slow down transaction processing. On the other hand, inefficient computation is a function of the overhead incurred by the algorithms executing operations and state transitions, which can include everything from simple transfers to complex smart contracts and signature verifications.

These bottlenecks are mutually reinforcing—delays in state access can prolong the time to computation, while inefficient computational practices can strain state management. Moreover, proposed improvements to addressing these issues often require systemic improvements such as sharding or adopting stateless architectures, which enhance state access and computation efficiency to improve execution performance.

Bottleneck #1: Inefficient State Access

The cost and speed required to access the state of a blockchain are critical bottlenecks to performant execution environments and can be distilled to the issue of state bloat

.

In blockchains, the state of the world is managed and updated through specific data structures calledtrees

. Trees are integral to blockchains, providing a secure and efficient way to give parties external to the executing node guarantees around the blockchain's correct state. Each update within a trie generates a new root hash, which light clients can reference to verify transactions and account balances without maintaining the entire chain.

Ethereum specifically relies on a data structure known as the Merkle Patricia trie (MPT)

, which comprises four sub-tries.

As Ethereum adds more smart contracts and tokens to its state, its state trie becomes larger and more complex. As the state grows, it requires more storage space, more computational resources to process, and more bandwidth to transmit. At the same time, the node’s hardware constraints stay roughly the same.

This state growth directly impacts Ethereum's performance because the state is stored in disk, and disk operations incur a high overhead. While accessing data from a CPU register can take 0.1 nanoseconds, it can take between 10 and 100 microseconds (100x–1000x slower) to access data from a disk, roughly translating to 200,000 CPU instructions that could have been executed in that time. That equates to a conservative estimate of 36 ERC-20 transfers that could have been!

Exacerbating this issue, blockchains have many inefficient access patterns for reading and writing to state. For example, the non-sequential structure of the Merkle Patricia trie inherently leads to these disk input/output (I/O)

operations reading from and writing to various unpredictable locations on the disk. The random nature of transaction inputs and the subsequent state changes they trigger lead to a scattered data access pattern which significantly slows the process of verifying and updating state and only utilizes a portion of a hardware device’s capacity.

All in all, the state management primitives for blockchains are far from achieving their absolute potential, and numerous advancements can be made to improve computational efficiency.

Bottleneck #2: Inefficient Computation

Execution layers also face the bottleneck of inefficient computation, which manifests in various ways.

For one, many process transactions sequentially, underutilizing modern multi-core processors capable of handling multiple operations simultaneously. This sequential execution leads to inevitable CPU idle times between transactions, wasting valuable computational resources.

Additionally, using virtual machines involves translating high-level smart contract operations into bytecode

—a lower-level, platform-independent code—which is then executed instruction by instruction. This translation and execution process introduces significant overhead, especially for applications with complex and frequently repeated application-specific tasks.

These inefficiencies lead to suboptimal utilization of computational resources and hinder the performance of execution layers.

Solutions: Inefficient State Access

There are a few distinct ways that teams are improving the rate at which state can be retrieved and updated from an executing node’s hardware, including simplifying complex data structures and finding ways to reduce the costly disk I/O operations that lead to state bloat.

Statelessness & In-Memory Computing

Some execution layers address state bloat by simply accepting it in the short run

. They shift state data storage from slower disk-based systems to faster random access memory (RAM). Accessing state information in RAM significantly reduces the overhead associated with disk operations, which are slower and more resource-intensive.

However, this approach challenges the core principle of decentralization. Storing increasingly large amounts of state data in RAM necessitates more advanced and expensive hardware, which could limit the ability of individuals to participate as node operators. Consequently, as the hardware requirements escalate, fewer entities can afford to run these nodes

To balance the attractiveness of computing in memory with trust minimization, both L1s (such as Ethereum) and L2s are pursuing a scalability roadmap that relies on unbundling the role of a validator into separate, centralized executing nodes with many verifying nodes. In this model, highly performant block producers with the hardware requirements to compute in memory are responsible for generating blocks, and cryptographic proofs (fraud and validity proofs) are leveraged by verifying nodes to keep block producers accountable.

As a result, these systems should allow block producers to maximize their speed because they can be expected to compute in memory, eliminating disk I/Os completely during execution. Because the latency of RAM is typically under 100 nanoseconds, the latency of state access is reduced by up to 1000x relative to disk-based implementations.

In parallel, fraud and validity proofs are used in place of decentralized consensus to scale the trust-minimization properties of the system along with its throughput. As a result, powerful, centralized block-producing nodes are counterbalanced by verifying nodes that can be run on much less expensive hardware. These nodes perform the critical function of independently verifying proofs of state transitions (or invalid state transitions) to maintain an accurate view of the state without the burden of storing the entire blockchain state.

To facilitate this process in a trust-minimized fashion, execution layers must implement a degree ofstatelessness

, the most popular being the concept of "weak statelessness." Weak statelessness is achieved by mandating that block producers provide a cryptographic attestation known as a witness to a verifying node. This witness encapsulates all the proposed state changes by the new block, enabling validators to verify these changes without additional historical data.

Although this concept can be applied using various tree structures, Verkle trees are often preferred to Merkle trees for their efficiency. Merkle trees require the inclusion of all sibling node hashes along the path from a data point (leaf) to the tree's root to prove data integrity. This requirement means that the size of the witness (the proof of integrity) grows with the tree's height, as each level necessitates additional hashes. Consequently, verifying data integrity in Merkle trees becomes computationally intensive and costly, especially for large datasets. In contrast, Verkle trees streamline this process, reducing the overhead associated with computation and storage in generating and verifying new blocks.

Verkle tree scaling from Inevitable Ethereum's "Verkle Tree"

Verkle trees enhance the structure of traditional Merkle trees by streamlining the connections between leaves and the root and eliminating the need to include sibling nodes in the verification process. In a Verkle tree, verifying a proof involves only the value at the leaf node, a commitment to the root node, and a single vector commitment based on polynomial commitments, which replaces the multiple hash-based commitments found in Merkle trees. This shift allows Verkle trees to maintain a fixed-size witness, which does not increase with the tree's height or the number of leaves verified, significantly improving the efficiency of storage and computation during data verification.

Over the coming years, we'll see implementations of statelessness happen at the L1 and L2 levels with varying configurations. According to the latest Ethereum roadmap, validators can rely on block builders to provide Verkle proofs regarding the state of certain blocks and verify these lightweight proofs instead of maintaining Ethereum's state directly.

At the L2 level, teams like MegaETH are actively applying the concept of statelessness to the design of optimistic rollups. In their design, the sequencer node generates a witness for each block containing the necessary state values and intermediate hashes while emitting a state delta representing the changes in the state. Verifier nodes can then re-execute any block by retrieving the witness from the DA layer or a peer-to-peer network without storing the entire state. In parallel, full nodes update their state by applying the state deltas disseminated through the network, allowing them to stay synchronized without re-executing transactions or storing the entire state history.

However, it's also worth pointing out that the benefits of statelessness and the resulting ability to compute in memory isn't a silver bullet for the performance of the execution layer.

Real-time TPS from MegaETH's "Understanding Ethereum Execution Layer Performance"

As co-founder of MegaETH, Yilong Li, identifies in the following research presentation on Ethereum execution, there are other inefficiencies to the data structures and access patterns onchain that remain optimized.

Improving the Databases

Teams working on execution layers are finding ways to improve the structure of these databases themselves to eliminate some of the bottlenecks experienced by Ethereum and other EVM-compatible blockchains in dealing with inefficient state access, which has a domino effect on computational efficiency.

In fact, the limitations of existing database designs found in the EVM informed Monad's* decision to go beyond purely optimizing for computational efficiency to achieve parallelization. Monad found that even after implementing parallel execution, they only saw a small speedup in performance because multithreaded read and write requests to the database blocked each other. As a result, Monad implemented a database compatible with asynchronous IO (AIO), or parallel access , as a critical part of the solution.

Async I/O

I/O operations—such as reading from or writing to storage devices—often create bottlenecks, particularly with mechanical hard disk drives (HDDs). These drives require the physical movement of a read/write head to access data, which can significantly slow down data processing.

AIO addresses this challenge by allowing programs to perform I/O operations concurrently with other processes. Essentially, a program can initiate an I/O operation and move on without waiting for it to complete. It does this by registering a callback function or a promise that the operating system or an I/O library will fulfill once the I/O operation finishes. This asynchronous approach allows the main program to continue executing other tasks, improving overall efficiency by not stalling for I/O tasks to complete.

Asynchronous I/O can be implemented with both traditional HDDs and solid-state drives (SSDs), although the benefits are more pronounced with SSDs. HDDs can perform AIO, but their mechanical nature means they're inherently slower than SSDs which store data on flash memory and have no moving parts, resulting in faster access times.

For instance, Monad utilizes a custom state backend optimized for SSD storage, which supports high levels of parallel data processing and reduces I/O latency. This setup is more efficient than systems relying solely on traditional disk-based storage or those using in-memory databases, which may still face delays from frequent data writes to and reads from slower storage media.

Similarly, Reth employs a method that separates database operations from the core EVM execution engine. This setup allows the EVM bytecode to execute sequentially on a single thread to maintain consistency while database I/O tasks are offloaded to parallel processes. Reth uses the actor model—a software architecture pattern—to manage these parallel processes effectively, ensuring that I/O operations do not interrupt the EVM interpreter.

State Merklization Frequency

Another vector for optimization is the state merklization frequency. Ethereum's current model of merklizing state after every block introduces significant overhead, requiring frequent writes to and reads from the disk and continuous trie traversals. Merkle trees usually work by grouping intermediate hashes into sets of 16 (called a node) and storing them in a key-value store database where the key is the node hash and the value is the node itself.

Traversing this tree to find and update data requires one random disk access for each layer of the tree to be traversed, and traversing a naive Merkle tree will require roughly eight sequential database queries per entry.

Solana's approach of updating the state commitment only at the end of each epoch allows for the amortization of write costs over many transactions within that period. If a state entry is modified multiple times within the same epoch, each write does not require an immediate update to the Merkle root. This reduces the overall computational overhead associated with state updates during the epoch. Consequently, the cost associated with reading from the state remains constant, or O(1), because the state can be read directly without needing to traverse a Merkle path each time.

Reducing the frequency of merklization in Ethereum could decrease the overhead from state reads and writes, enhancing performance. However, light clients would need to replay block changes to track state between epochs or submit onchain transactions for state verification, and such a change is not currently compatible with Ethereum.

Efficient & Specialized Data Structures

Moreover, layered tree structures within existing Ethereum clients generally cause inefficient state access patterns, further contributing to state bloat. While Ethereum's state is structured as an MPT, it is also then stored in Ethereum client databases such as LevelDB, PebbleDB (utilized by go-ethereum), or MDBX (employed by Erigon) that store data in Merkle trees such as a B-Tree or LSM-Tree.

In this setup, a data structure is rooted into another data structure of a separate type, creating "read amplification" from navigating internal tree structures atop clients that operate under another Merkle tree-based system. Read amplification can be understood as the result of the multiple steps for accessing or updating information contained within a state, which requires navigating the outer tree to find the entry point into the MPT before executing the required operation. As a result, the number of disk accesses for a random read is multiplied by a log(n) factor.

To solve this, Monad natively harnesses a Patricia trie data structure on disk and in memory. From a technical perspective, Patricia tries are often superior to other Merkle tree structures due to their unique combination of space efficiency, efficient prefix matching, and minimal node traversal. The trie's design collapses nodes with single children and streamlines lookups, inserts, and deletions, reducing the number of disks or network I/O operations required. Moreover, a Patricia trie's adeptness at handling prefix matching enhances performance in applications needing rapid partial key searches.

Another bottleneck specific to tree-based structures is that accessing or updating data requires traversing multiple layers, leading to numerous sequential disk accesses. Sovereign Labs addresses this inefficiency by advocating for a binary Merkle tree configuration. This pivotal shift to a binary structure drastically reduces the number of potential paths during tree traversal, directly reducing the hash computations needed for updates, insertions, and cryptographic proofs.

Binary Merkle tree configuration from Sovereign Labs's "Nearly Optimal State Merklization"

An additional example in this category is the Reth team configuring Reth to pre-fetch intermediate trie nodes from the disk during execution by notifying the state root service about storage slots and accounts touched.

State Expiry

State expiry is a mechanism to manage and reduce the size of the blockchain state by removing data that hasn't been accessed for a set period of time. While expiry is often bucketed under the "statelessness" category, it's critical to distinguish these concepts in the context of execution.

Statelessness improves execution by increasing an executing node's ability to compute in memory, but the improvements to execution stem from the beefier hardware requirements across fewer nodes that execute transactions. In contrast, state expiry can be applied to blockchains with both few and many executing nodes.

There are a couple of methods commonly discussed for implementing state expiry:

- Expiry by Rent: This method involves charging a maintenance fee, or "rent," to keep accounts active within the state database. If the rent isn't paid, the accounts are archived until a fee is paid to restore them.
- Expiry by Time: Here, accounts are deemed inactive if they haven't been accessed—meaning no transactions or interactions—for a specified duration.

Expiry by Rent: This method involves charging a maintenance fee, or "rent," to keep accounts active within the state database. If the rent isn't paid, the accounts are archived until a fee is paid to restore them.

Expiry by Time: Here, accounts are deemed inactive if they haven't been accessed—meaning no transactions or interactions—for a specified duration.

Both methods aim to maintain only actively used data in the immediate, accessible state while pushing out the older, less frequently accessed data to an archived state that does not burden the main system.

By maintaining a smaller and more manageable state, state expiry reduces the "state bloat" that can severely hinder blockchain performance. A smaller state size allows nodes to quickly navigate and update the state, translating to faster execution because nodes spend less time scanning and more time processing.

Execution Sharding

Sharding optimizes resource utilization and performance by distributing tasks and data across a limited number of specialized nodes (not every node executes a global state).

In a sharded blockchain architecture, the global state is divided into distinct partitions called shards

. Each shard maintains its portion of the state and is responsible for processing a subset of the network's transactions. Transactions are assigned to specific shards based on a deterministic sharding function, which considers various factors such as the sender's address, the recipient's address, and the hash of the transaction data. This minimizes the need for cross-shard communication and enables more efficient transaction execution.

Sharding diagram from Vitalik's "The Limits to Blockchain Scalability"

This becomes evident when exploring NEAR Protocol's sharding design, Nightshade, which achieves statelessness to scale sharding without compromising trust minimization.

In Nightshade, the blockchain is structured as a single logical chain, with each block composed of multiple "chunks" and one chunk being allocated per shard. These chunks contain the transactions and state transitions specific to each shard. Including chunks from all shards within a single block allows for a unified view of the entire blockchain state and simplifies the process of cross-shard communication.

Similarly to proper-builder separation (PBS)

on Ethereum, Nightshade explicitly delineates the roles of stateful and stateless nodes. On NEAR, stateful validators are assigned to specific shards and are responsible for collecting transactions, executing them, and producing shard-specific chunks. They maintain the full state of their assigned shard and generate state witnesses for validators to use during the validation process.

Meanwhile, stateless validators are randomly assigned to validate specific shards on a per-block basis. They do not need to maintain the full-sharded state and rely on state witnesses provided by the block producers from other shards to validate the state transitions and transactions within a chunk. The random assignment of validators to shards helps ensure the network's security and integrity, as it makes it more difficult for malicious actors to collude and control a specific shard.

Since each node in the network only needs to handle the data for its respective shard rather than the entire network's data, the storage and computational burden on individual nodes is reduced.

Solutions: Inefficient Computation

Parallelizing Execution

Time to address the elephant in the room: parallelization

. Parallelizing transaction execution enables processing multiple transactions by utilizing multiple computing resources concurrently. This allows for increased throughput as hardware resources are scaled up during periods of high demand.

However, it's important to consider that multiple execution components can be parallelized, many of which are implemented by coprocessors such as Lagrange* and alternative blockchain clients such as Firedancer to improve blockchains' performance significantly. Specifically, parallelization can involve:

1. Parallelizing State Access
2. Parallelizing Specific Operations
3. Parallelizing Consensus and Execution

Parallelizing State Access

Parallelizing Specific Operations

Parallelizing Consensus and Execution

Parallelizing State Access

Parallelizing state access brings two critical benefits:

1. Parallel EVMs distribute transaction processing across several CPU cores. This setup allows multiple transactions to be handled concurrently rather than forcing them to queue up for a single resource.
2. When a transaction waits for data from storage—which can introduce significant latency—the system doesn't remain idle. Instead, it can switch to another transaction that is ready to execute. This is possible because multiple cores can handle different tasks independently and simultaneously.

Parallel EVMs distribute transaction processing across several CPU cores. This setup allows multiple transactions to be handled concurrently rather than forcing them to queue up for a single resource.

When a transaction waits for data from storage—which can introduce significant latency—the system doesn't remain idle. Instead, it can switch to another transaction that is ready to execute. This is possible because multiple cores can handle different tasks independently and simultaneously.

The primary challenge in parallelizing transaction execution stems from managing concurrent access to the shared global state without violating the ACID rules for updating distributed systems. If a blockchain has a bunch of transactions executing in parallel, some of them will conflict. As a result, the two primary methodologies for parallelizing state access differ on when

they dedicate resources to resolving conflicts: the pessimistic execution

(or memory lock) model and the optimistic execution

model.

Pessimistic Execution

The pessimistic execution model is a transaction processing approach requiring transactions to declare the state variables they will access (read or write) during execution. This information is included in the transaction's metadata, allowing the runtime to analyze the access patterns before execution.

By examining the read-write access patterns, the runtime can identify transactions with non-overlapping access sets, enabling parallel execution of non-overlapping and read-only transactions and improving throughput. The runtime creates parallel transaction queues for each CPU thread on a validator node, ensuring that transactions with non-conflicting access patterns are processed concurrently.

As a result of this design choice, the pessimistic execution model benefits from fine-grained control over resource allocation, allowing for segmenting or partitioning of a blockchain's state space.

Parallelization effectively creates multiple, synchronously composable independent execution shards underpinned by a unified security model. It helps address network congestion and optimize gas costs through precise resource management and dynamic fee markets. By identifying state-access "hotspots" (areas of high transactional demand), the system can implement targeted optimizations like differentiated fee pricing, rate limiting, or allocating additional resources to high-contention states. It's important to note that Solana's current implementation of parallelization does not fully realize the potential of localized fee markets

To ensure data consistency in concurrent access, the pessimistic execution model utilizes a locking mechanism

. Before a transaction can access a specific state variable, it must acquire a lock on that variable. The lock provides the transaction with exclusive access to the variable, preventing other transactions from modifying it simultaneously. The lock is released once the transaction is executed, allowing other transactions to access the variable.

In Solana's Sealevel runtime, which implements this pessimistic execution model, transactions specify the accounts they will read or write during execution. Sealevel analyzes the access patterns and constructs parallel transaction queues for each CPU thread on a validator node. If an account is accessed multiple times, it is listed sequentially in a single queue to prevent conflicts. Transactions not processed within the leader node's block time are bundled and forwarded to the next scheduled leader for processing.

Unspent transaction output (UTXO)-based

systems improve computational efficiency similarly. UTXOs involve specific units of currency—UTXOs—associated with an individual's wallet. For each of said wallet's transactions, UTXOs are expended and replaced with new ones; one or more UTXOs are created for the receiver, representing the payment, and another is typically created for the initiator, representing any change due back.

By defining which contracts will be touched, transactions that touch disjoint sets of contracts can be executed in parallel by executing nodes (which can be accomplished in the "accounts": data model with strict access lists). However, to gain compatibility with Ethereum-style smart contracts, UTXO schemes such as Fuel's constrain block-producing nodes to execute transactions with overlapping access lists sequentially.

Nevertheless, the pessimistic execution model has limitations. Transactions must accurately declare their access patterns upfront, which can be challenging for complex or dynamic transactions where the access patterns may depend on input data or conditional logic. Inaccurate or incomplete access pattern declarations can cause suboptimal performance and potential runtime errors. Additionally, the locking mechanism can introduce latency and reduce concurrency when many transactions compete for the same state variables. This contention can form performance bottlenecks, as transactions may spend a significant portion of their execution time waiting to acquire locks on high-demand state variables.

More importantly, this model places a considerable burden on developers, who must have a deep understanding of their contracts' data dependencies to specify necessary state accesses beforehand accurately. This complexity can introduce challenges, especially in designing applications with dynamic and complex state interactions, such as decentralized exchanges or automated market makers.

Optimistic Execution

In contrast, the optimistic execution model

adopts a "speculative" approach to transaction execution, allowing transactions to execute in parallel without needing upfront state access declarations.

Instead of preventing conflicts before they happen, transactions are optimistically executed in parallel, assuming they are independent. The runtime employs techniques like [multi-version concurrency control](https://en.wikipedia.org/wiki/Multiversion_concurrency_control#:~:text=Multiversion%20concurrency%20control%20(MCC%20or,languages%20to%20implement%20transactional%20memory.) (MVCC) and software transactional memory (STM) to track read and write sets during execution. After execution, the runtime detects any conflicts or dependencies. It takes corrective measures, such as aborting and re-executing conflicting transactions, but can do so by reading from memory instead of disk to identify conflicting transactions.

The optimistic execution model simplifies the development process, allowing programmers to focus on writing contract logic without worrying about declaring state access patterns. Because transactions do not need to declare their state interactions upfront, developers are afforded more freedom in designing their smart contracts, allowing for more complex and dynamic interactions with the blockchain's state. The optimistic execution model is particularly well suited for platforms that support a high volume of transactions and complex dapps, as it can offer higher throughput and scalability than the pessimistic model.

One notable implementation of this model is found inAptos and the MoveVM of Movement Labs*, which employs a technique known as Block-STM. In Block-STM, transactions are first executed in parallel; then, conflicting transactions are identified and scheduled for re-execution based on the detected dependencies. This approach ensures processing resources are continuously utilized, improving throughput while maintaining the integrity of the transactional workflow.

Aptos's Block-STM from "Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing"

Despite its advantages, the optimistic execution model also comes with challenges. The need for runtime conflict detection and the possibility of transaction aborts and retries introduce computational overhead and complexity. In addition, maintaining multiple versions of the state and managing the overhead associated with conflict resolution requires sophisticated system design and robust concurrency control mechanisms to ensure the blockchain's integrity and performance.

Block-STM leverages MVCC to effectively manage concurrent writes and maintain multiple versions of data, thereby preventing conflicts between simultaneous write operations. It incorporates a collaborative scheduler to coordinate the execution and validation tasks across multiple threads, ensuring that transactions are committed in the order they were started. This setup minimizes transaction aborts by using dynamic dependency estimation, which allows transactions with dependencies to efficiently wait and resolve these dependencies before proceeding.

Additionally, the account model used by MoveVM differs from that of Ethereum's EVM, which leads to fewer collisions. In Ethereum, a token is typically managed by a single smart contract, potentially causing multiple token transactions to interact through the same contract address, increasing the likelihood of conflicts. In contrast, MoveVM assigns tokens to individual user accounts, reducing the chance of such conflicts as each transaction usually interacts with different account addresses.

In Monad, the initial set of transactions executed in parallel can be framed as an I/O phase, which may produce immediately-committable results, and the following "retry" phase, which requires a small amount of work to clear conflicting remaining transactions. These conflicting transitions are surfaced and pulled into cache, allowing for execution overhead to be reduced because they live in memory. While most state lives on disk, conflicting transactions are accessed quickly at execution time.

The pessimistic and optimistic execution models offer distinct approaches to handling transaction execution and state management in blockchains. The choice between these models involves tradeoffs between upfront complexity in state access specification and the computational overhead associated with dynamic conflict resolution.

Data & Task Parallelism

Data and task parallelism focus on optimizing performance by distributing computational loads across multiple processors: data parallelism segments a dataset for simultaneous processing, while task parallelism assigns different tasks to various processors to operate concurrently.

These optimizations are distinct but interdependent with state access parallelism, which manages and synchronizes access to shared resources like memory or databases to prevent conflicts and ensure data integrity when multiple processes or threads operate simultaneously.

Data Parallelism

Data parallelism involves parallelizing specific operations across multiple data elements simultaneously. This approach is particularly beneficial when the same operation needs to be applied to a large dataset or when performing computationally intensive operations on multiple input values. The key unlock comes from distributing the data across multiple processing units and executing the same operation concurrently on different data elements.

One common technique for data parallelism issingle-instruction, multiple data (SIMD)

, which allows a single instruction to be executed simultaneously on multiple data elements. Modern CPUs often have built-in SIMD capabilities, enabling them to perform parallel operations on multiple data points. By leveraging SIMD instructions, developers can achieve significant speedups for certain types of operations, such as mathematical computations, data transformations, or signal processing.

For example, consider a scenario where you must apply a complex mathematical function to a large array of numbers. Instead of processing each number sequentially, SIMD can operate on multiple numbers simultaneously. This simultaneous processing is achieved by loading a subset of the numbers into the CPU's SIMD registers, executing the mathematical function on all the loaded numbers in parallel, and then storing the results back into memory. By processing multiple numbers at once, SIMD can greatly reduce the overall execution time.

Firedancer's work on ED25519 signature verification demonstrates the power of SIMD for optimizing complex computations. The signature verification process involves arithmetic operations within Galois Fields, which can be computationally intensive. By leveraging SIMD instructions, Firedancer can perform these operations on multiple data elements concurrently, resulting in significant performance improvements. These optimizations will be critical in improving the performance of Solana, which has already implemented parallelization of state access.

Task Parallelism

Task parallelism involves parallelizing different tasks or operations within a program across multiple processing units. This approach is useful when a program consists of multiple independent tasks that can be performed concurrently. By assigning each task to a separate processing unit, such as a CPU core or a GPU, the overall execution time can be reduced.

Task parallelism is commonly used in scenarios where a program needs to perform multiple complex operations simultaneously. For instance, consider a video processing application that needs to apply different filters and effects to a video stream in real-time. Instead of using every compute unit to collectively apply each filter sequentially, task parallelism can distribute the workload across the multiple processing units. One processing unit can be responsible for applying a blur filter while another unit applies a color correction filter, and so on. By executing these tasks in parallel, the application can achieve faster processing and maintain a smooth user experience.

Lagrange's ZK MapReduce (ZKMR) leverages data and task parallelism to efficiently parallelize and generate proofs of distributed computations on large datasets. In the map phase, the input dataset is partitioned into smaller chunks, and each chunk is processed independently by a separate mapper worker or machine in parallel (task parallelism). The "map" operation can be parallelized within each mapper task across multiple cores or processors (data parallelism). Similarly, in the reduce phase, the "reduce" operation on the values associated with each key can be parallelized within each reducer task (data parallelism). In contrast, the reducer tasks are executed parallel across multiple workers (task parallelism).

By combining data parallelism and task parallelism, ZKMR can achieve efficient scaling and performance for complex computations on massive datasets while maintaining zero-knowledge guarantees through recursive proof composition.

Verifying an arbitrary MapReduce procedure in ZK from Lagrange's “Introducing ZK MapReduce”

Lagrange’s ability to generate storage proofs for SQL computations over888,888 storage slots in 1 minute and 20 seconds demonstrates the power of ZKMR, as well as the task and data parallelism that underpin it. Moreover, Lagrange’s recent Reckle Trees paper underscores the need for parallelism in that it ensures that batch proofs of onchain data are also computable in $(\log n)$, independent of the batch size.

Parallelizing Consensus & Execution

While this piece doesn't address consensus, blockchains can also parallelize the process of consensus and execution. Traditional blockchains often process transactions sequentially, reaching a consensus on a block's transactions (block N) before executing them. Parallel processing of consensus and execution phases significantly boosts execution efficiency and is a technique exemplified by systems like Monad. As the network reaches consensus for block N, it concurrently executes transactions for the previous block (N-1).

This strategy ensures continuous, efficient use of computational resources, effectively reducing idle times and enhancing the network's ability to process transactions quickly. These improvements increase the system's throughput and the cost of capital required to spam the network.

Interpreters & Reducing Overhead

When smart contracts are written in languages like Solidity, they are first compiled into the lower-level bytecode. Then, the EVM uses an interpreter to execute this bytecode. The interpreter reads and executes each instruction sequentially, akin to translating a foreign language in real-time as it is being spoken. Paradigm’s latest piece on Reth points out that this leads to overhead since each instruction must be processed individually and converted from bytecode to machine instructions during runtime.

Reth is addressing EVM inefficiencies by incorporating a just-in-time (JIT) compiler

. This compiler translates bytecode into native machine code shortly before execution, circumventing the resource-intensive interpretation process typically required during runtime.

The Reth article mentions that 50% of EVM execution time under an interpreter-based system is dedicated to processes that JIT could theoretically optimize, suggesting the possibility of doubling execution speed with JIT implementation. However, as Yilong points out in this presentation, while JIT can significantly decrease the time needed for processing specific opcodes, it may not drastically impact overall execution. This is because a substantial portion of the 50% of EVM execution times that JIT takes up involves "host" and "system" operations (Slide 13), which are not amenable to JIT optimizations like "arithmetic" or "control" due to their non-computational nature.

While interpreters may limit performance, they do create the opportunity for “translation,” which increases the scope of code that can leverage new virtual machines, lowering the overhead for developers to use designer blockspace. For example, Movement Labs has developed Fractal, enabling developers to deploy their Solidity-based contracts on MoveVM. Fractal works by compiling Solidity into an Intermediate Language containing instructions articulated in EVM opcodes, which are then mapped to their MoveVM bytecode counterparts, allowing Solidity contracts to run in the MoveVM environment.

Specialized & Customized State Machines

Customizing the execution layer involves designing specialized state machines optimized for specific applications. Not only does this mean that an execution environment can forgo the need for a virtual machine entirely, but it also enables applications to tailor the instruction set architecture (ISA), data structures, and execution model to their specific needs. The key performance benefit of tailoring an ISA to a specific application comes from reducing the overhead of translating the application's computational patterns into the general-purpose instructions of a traditional ISA. General-purpose CPUs use basic instruction sets (i.e., add, load, branch) to support running different types of software. However, when applications frequently repeat the same multistep operations, implementing these patterns using sequences of simple instructions becomes inefficient.

For example, database applications may need to constantly traverse tree data structures, look up entries, update values, and rebalance trees. On a normal CPU, mapping these higher-level operations requires breaking them into long sequences of low-level micro-ops, such as loads, stores, branches, and arithmetic, executing individually on the general hardware. In contrast, an ISA customized for databases can fuse these recurring patterns into optimized wider instructions that leverage specialized hardware. A "TraverseTree" instruction could calculate memory addresses, load relevant nodes, and compare keys using parallel comparison circuits designed for that operation. "UpdateEntry" could directly gather the entry from the optimized database storage layout, modify it, and commit the new state all in a single instruction.

This eliminates redundant overhead from translating high-level operations down to simple instructions. It also allows the hardware to optimally execute the application using fewer but wider, explicitly parallel instructions tailored precisely to its needs.

LayerN’s Nord demonstrates the performance benefits of specializing execution environments and data structures through their specific use case of a verifiable order book. LayerN's approach focuses on optimizing the placement of trades into the order book data structure, while their pipelining mechanism is designed to efficiently insert new orders into the appropriate position within the order book's data tree. By tailoring the data structure and insertion algorithm to the specific requirements of an order book, LayerN achieves low-latency order placement and high throughput.

Alternatively, it's possible to lean into general-purpose execution environments that enable arbitrarily programmable modules that apps can plug into to optimize their performance. This approach prioritizes the developer experience over raw performance.

Fluent and CWD utilize a strategy that balances the tradeoffs between optimizing for raw computational performance and enhancing the developer experience and compatibility of the ecosystem. This approach centers on using WebAssembly (Wasm) as the VM to execute code. Wasm has become a preferred choice in web development due to its broad language support and the wide degree to which it has been adopted.

A developer's decision to use Wasm rather than native client execution reflects a strategic preference for the versatility and broad accessibility of a general-purpose execution environment. Although native execution, which runs code directly on hardware without a virtual machine, can offer better performance, it restricts cross-platform compatibility and is less accessible to developers. In contrast, Wasm ensures a uniform and secure execution environment across different platforms despite not achieving the same raw speed as native execution. This tradeoff aligns with Fluent and CWD's design philosophies, prioritizing developer productivity and broader ecosystem integration over maximum performance efficiency.

CosmWasm deployment (CWD), in particular, exemplifies this approach by not just employing Wasm for smart contract execution but also incorporating it into a more extensive framework designed to support the intricacies of blockchain operations. Enriched with "periphery logic," this framework offers advanced account management, a customizable gas mechanism, and optimized transaction ordering. These features contribute to a flexible, efficient, secure development environment that empowers developers to build scalable and complex dapps relatively easily.

Stackr* takes a different approach by combining the benefits of customized execution environments with the flexibility of traditional smart contract platforms. Stackr allows developers to code applications as rollups, enabling them to define their own rules for transaction ordering, execution, and configuration. In the Stackr model, developers can choose the ISA, data structures, and execution model that best suit their application's requirements.

Stackr’s micro-rollup design from “Introducing the Stackr SDK”

With Stackr, developers can apply state transition rules directly in the application's runtime rather than being constrained by the rules of a general-purpose VM, giving them the ability to streamline their instruction set to be more efficient and redefine the set of things that can be done in a runtime environment.

This results in more lightweight and efficient execution, as the business logic is implemented at the client level, eliminating the need for costly smart contract invocations and validation. As a result, the possibilities around how an application is configured expand in terms of the different types of languages, data structures, and signatures developers can use for a single app without sacrificing performance.

Conclusion

There are multiple paths to optimal execution layer performance.

No singular optimization to state access or parallelization stands out as a proprietary point of technical differentiation between execution layers when attempting to capture dapps. As we went through, the benefits of resource-based parallelization on Solana can be equally applied to Fuel's UTXO model. Anyone can use Amazon's insightful solutions to improve horizontal scalability through sharding and improve the execution layer performance.

While execution layer performance is a critical vector for winning over builders of decentralized applications, new L1s and L2s centered around improving execution must compete on other variables, including security, interoperability, and compatibility with existing tooling. For this reason, the proliferation of new interoperability layers—from Nebra to Statenet to Polygon's AggLayer—will be critical to developers buying designer blockspace, as they can build or buy specialized blockspace without sacrificing the synchronous composability and shared liquidity of traditional, general-purpose L1s.

Improvements to state management and computational efficiency are interdependent.

Across the communities designing new execution layers, the parallelization of state access has become a defining meme for the performance improvements they promise to bring. While this is for a good reason, as it could lead to a 5x improvement in the execution of the EVM evidence from Monad's early experimentation with parallelization demonstrates that its role is overemphasized if other improvements, such as async I/O, aren't developed in tandem.

Based on this, we can conclude that computational efficiency is often only achieved when we improve how state is accessed and stored. Efficient state management reduces the time and resources needed to access and manipulate data, which speeds up processing and reduces computational load.

Taking this a step further, incumbents may be making path-dependent choices that hinder their ability to compete with new blockchain designs that re-architect how state is managed and updated, given the inertia that a hard fork entails. As a result, specialized, modular execution layers and alternative L1s may be able to create defensibility around design choices for more efficient state storage and the protocols for reading from and writing to it. These design decisions offer a competitive advantage, as incumbents may encounter inertia in updating their database structures without a hard fork.

At the end of the day, a blockspace's values impact the design space for execution layers.

In understanding how we can improve execution layers, we can now delineate that the classes of optimizations differ according to two critical design choices—who

is executing transactions, and how many

nodes need to be involved? The techniques available to developers for solving execution bottlenecks differ significantly depending on a team's initial answers to these questions.

On one hand, monolithic L1s like Solana and Monad don't accept separating the validator role into heterogeneous powerful and weak nodes to accelerate performance. "Accepting" state bloat in the short-term isn't a viable solution, so they lean on improvements at the database layer and other components of the block production engine, such as consensus, to make up for the broader number of executing nodes deemed as a critical component and core value of the network. Because the security models of these L1s rely on the consensus of a more distributed set of validators with weaker hardware requirements, their data needs to be written to a database that lives on a disk, which is necessarily cheaper for a permissionless and maximally decentralized blockchain.

On the other hand, projects like Ethereum and its L2s are pursuing a roadmap that leans into centralization across their executing nodes through centralized block builders held accountable to weaker verifying proposer nodes through fraud or validity proofs.

Suppose centralized "executors" of transactions and state transitions are considered acceptable in pursuing a decentralized future. In that case, the law of physics states that systems that can 1) add blocks to a chain without requiring multiple actors to re-execute transactions, 2) increase validator requirements to maximize in-memory computation (and ignore the state bloat problem), and 3) reduce latency and consensus bottlenecks clearly win out compared to systems relying on extensive decentralization and consensus among nodes.

In seeking a balance between scalability and trust minimization, it's becoming apparent that the objective for execution layers should not be to optimize for decentralization blindly, nor must execution always be completely permissionless.

As we develop and implement a broader array of cryptographic tools, such as validity and fraud proofs, we effectively reduce the number of nodes necessary to resist censorship and maintain safety and liveness. This approach, however, involves tradeoffs, potentially impacting censorship resistance, ordering integrity, and liveness guarantees due to the possible centralization of executors.

As noted by Sreeram, the "minimum viable decentralization" does not mean that "validation should be permissionless" but that it should "just be rightly incentivized." This implies that a well-monitored system, where validators face significant repercussions for misconduct, can maintain safety and liveness without the need for excessive decentralization (h/t Sreeram).

Such governance models are already being tested in practical applications. For instance, rollups like Arbitrum are exploring governance or committee-based systems to enforce transaction ordering and leader selection rules, and they are considering mechanisms where sequencers use onchain data to uphold transaction ordering policies.

Despite these advancements, there is no definitive "pareto optimal frontier" for balancing decentralization with performance.

Ideological and technical considerations continue to favor decentralizing executing nodes to validate the state. While centralizing nodes reduces consensus overhead and upgrading hardware can significantly enhance performance, it remains to be seen whether these optimizations will attract developers focused on creating censorship-resistant applications and to what extent censorship resistance remains a core value in the industry.

*denotes an Archetype portfolio company

Special thanks to all the people in the arena for the thoughtful conversations and feedback that went into putting this piece togetherAsh Egan, Dmitriy Berenzon, Dima Romanov, Ismael Hishon-Rezaizadeh, Kautuk Kundan, Katie Chiou, Keone Hon, Larry0x, Movement Research Labs, Rushi Manche, Soumya Basu, Tyler Gehringer, Will Kantaros, and Yilong Li.