Privacy Policy

Last updated: July 25, 2023

General

Monad Labs, Inc. (“Company”) collects personal data as part of its products and services (“Services”). Your privacy and the privacy of all our users is very important to us. Please read this Privacy Policy to learn how we treat your

personal data.By using or accessing our Services in any manner, you acknowledge that you accept the practices and policies outlined below, and you hereby consent that we will collect, use, and share your information as described in this Privacy Policy.

What this Privacy Policy Covers

This Privacy Policy covers how we treat Personal Data that we gather when you access or use our Services. "Personal Data" means any information that identifies or relates to a particular individual and includes information referred to as "personally identifiable information" or "personal information" under applicable data privacy laws, rules or regulations. This Privacy Policy does not cover the practices of companies we don't own or control or people we don't manage.

Personal Data

Categories and Examples of Personal Data that are collected

- Web Analytics
  - Web page interactions
  - Non-identifiable request IDs
  - IP addresses
- Device IP/Data
  - Device information: type of device/operating system/browser used to access the Services
  - Domain server
- Other Identifying Information that You Voluntarily Choose to Provide
  - Identifying information submitted by you in emails, messages, or other content you upload to the Services
  - Employment, company, title, experience, education information that is self-provided by you
  - KYC data including name, citizenship, and other Personal Data necessary to perform KYC check

Categories of Third Parties with Whom We Share this Personal Data

- Service providers
- Affiliates
- Parties you authorize, access, or authenticate

Categories of Third Parties with Whom We Share this Personal Data

We collect Personal Data about you from the following categories of sources:

- You
  - When you provide such information directly to us
  - When you voluntarily provide information in free-form text through responses to surveys, applications, or questionnaires
  - When you send us an email or otherwise contact us
  - When you use the Services and such information is collected automatically
  - Through Cookies (defined in the "Tracking Tools and Opt-Out" section below).

Commercial or Business Purposes for Collecting Personal Data

- Providing, Customizing, and Improving the Services
  - Meeting or fulfilling the reason you provided the information to us
  - Providing support and assistance for the Services
  - Improving the Services including testing, research, internal analytics, and product development
  - Personalizing the Services and communications based on your preferences
  - Doing fraud protection, security, and debugging
  - Carrying out other business purposes stated when collecting your Personal Data or as otherwise set forth in applicable data privacy laws

- Corresponding with You
  - Responding to correspondence that we receive from you
  - Contacting you when necessary or requested, and sending you information about Monad or the Services
  - Sending emails and other communications based on your preferencesCorresponding with You

- Meeting Legal Requirements and Enforcing Legal Terms
  - Fulfilling our legal obligations under applicable law, regulation, court order or other legal process such as preventing, detecting, and investigating security incidents and potentially illegal or prohibited activities
  - Protecting the rights, property or safety of you, Monad or another party
  - Enforcing any agreements with you
  - Responding to claims that any posting or other content violates third-party rights
  - Resolving disputes

The examples provided are not exhaustive lists of collected data. However, we will not collect additional categories of Personal Data or use the Personal Data we collected for materially different, unrelated or incompatible purposes without providing notice or updating the Privacy Policy.

Disclosure of Personal Data

We disclose your Personal Data to the categories of service providers and other parties listed in this section.

- Service Providers
  These parties help us provide the Services or perform business functions on our behalf. They include:
  - Hosting, technology, and communication providers
  - Security and fraud prevention consultants
  - Support and customer service vendors

- Analytics Partners
  These parties provide analytics on web traffic or usage of the Services. They include:
  - Companies that track how users found or were referred to the Services
  - Companies that track how users interact with the Services

- Parties You Authorize, Access, or Authenticate
  - Third parties you access through the services
  - Social media services

Legal Obligations

We may share any Personal Data that we collect with third parties in conjunction with any of the activities set forth under "Meeting Legal Requirements and Enforcing Legal Terms" in the "Our Commercial or Business Purposes for Collecting Personal Data" section above.

Business Transfers

All of your Personal Data that we collect may be transferred to a third party if we undergo a merger, acquisition, bankruptcy or other transaction in which that third party assumes control of our business (in whole or in part). Should one of these events occur, we will make reasonable efforts to notify you before your information becomes subject to different privacy and security policies and practices.

Non-Personal Data

We may create aggregated, de-identified or anonymized data from the Personal Data we collect, including by removing information that makes the data personally identifiable to a particular user. We may use such aggregated, de-identified or anonymized data and share it with third parties for our lawful business purposes, including to analyze, build and improve the Services and promote our business, provided that we will not share such data in a manner that could identify you.

Tracking Tools and Opt-Out

The Services use cookies and similar technologies such as JavaScript (collectively, "Cookies") to operate and improve our Services. Cookies are small pieces of data – usually text files – placed on your computer, tablet, phone or similar device when you use that device to access our Services.

We use the following types of Cookies:

- Performance/Analytical Cookies.
  These Cookies allow us to understand how visitors use our Services. They do this by collecting information about the number of visitors to the Services, what pages visitors view on our Services and how long visitors are viewing pages on the Services. For example, Google LLC ("Google") uses cookies in connection with its Google Analytics services. Google's ability to use and share information collected by Google Analytics about your visits to the Services is subject to the Google Analytics Terms of Use and the Google Privacy Policy. You have the option to opt-out of Google's use of Cookies by visiting the Google advertising opt-out page at http://www.google.com/privacy_ads.html or the Google Analytics Opt-out Browser Add-on at https://tools.google.com/dlpage/gaoptout/.

You can decide whether or not to accept Cookies through your internet browser's settings. Most browsers have an option for turning off the Cookie feature, which will prevent your browser from accepting new Cookies, as well as (depending on the sophistication of your browser software) allow you to decide on acceptance of each new Cookie in a variety of ways. You can also delete all Cookies that are already on your device. If you do this, however, you may have to manually adjust some preferences every time you visit our website and some of the Services and functionalities may not work.

To explore what Cookie settings are available to you, look in the "preferences" or "options" section of your browser's menu. To find out more information about Cookies, including information about how to manage and delete Cookies, please visit http://www.allaboutcookies.org/.

Data and Security Retention

We seek to protect your Personal Data from unauthorized access, use and disclosure using appropriate physical, technical, organizational and administrative security measures based on the type of Personal Data and how we are processing that data. Although we work to protect the security of data that we hold in our records, please be aware that no method of transmitting data over the internet or storing data is completely secure.

We retain Personal Data about you for as long as we have a business purpose to do so or as otherwise necessary to provide you with our Services. In some cases we retain Personal Data for longer, if doing so is necessary to comply with our legal obligations, resolve disputes or collect fees owed, or is otherwise permitted or required by applicable law, rule or regulation. We may further retain information in an anonymous or aggregated form where that information would not identify you personally.

Child Privacy

Our services are not directed to children under 16 years of age, and we do not knowingly collect or solicit Personal Data about children under 16 years of age. If you are a child under the age of 16, please do not attempt to register for or otherwise use the Services or send us any Personal Data. If we learn we have collected Personal Data from a child under 16 years of age, we will delete that information as quickly as possible. If you believe that a child under 16 years of age may have provided Personal Data to us, please contact us.

State Law Privacy Rights

California

Under California Civil Code Sections 1798.83-1798.84, California residents are entitled to contact us to prevent disclosure of Personal Data to third parties for such third parties' direct marketing purposes; in order to submit such a request, please contact us.

Changes to this Privacy Policy

We reserve the right to modify this Privacy Policy at any time in our sole discretion. If we make any significant changes to this policy, we will alert you to any such changes by placing a notice on the Monad website, by sending you an email and/or by some other means. Please note that if you've opted not to receive legal notice emails from us (or you haven't provided us with your email address), those legal notices will still govern your use of the Services, and you are still responsible for reading and understanding them. If you use the Services after any changes to the Privacy Policy have been posted, that means you agree to all of the changes. Use of information we collect is subject to the Privacy Policy in effect at the time such information is collected.

Contact

If you have any questions or comments about this Privacy Policy, the ways in which we collect and use your Personal Data or your choices and rights regarding such collection and use, please do not hesitate to contact us.

Email: support-wpp@monad.xyz

Join the Community

News, Press, Media
Privacy Policy
Branding & Media Kit

Our Newsletter

Drop your email to get the latest on Monad happenings, directly from the source. No fomo.

email address

SUBSCRIBE

Extreme Parallelized Performance

Superscalar Pipelining for the EVM

Monad is a decentralized, developer-forward Layer 1 smart contract platform that ushers in a new paradigm of possibility through pipelined execution of Ethereum transactions.

Crafted for productivity, fundamentally optimized.

JOIN THE COMMUNITY

10,000
transactions per second

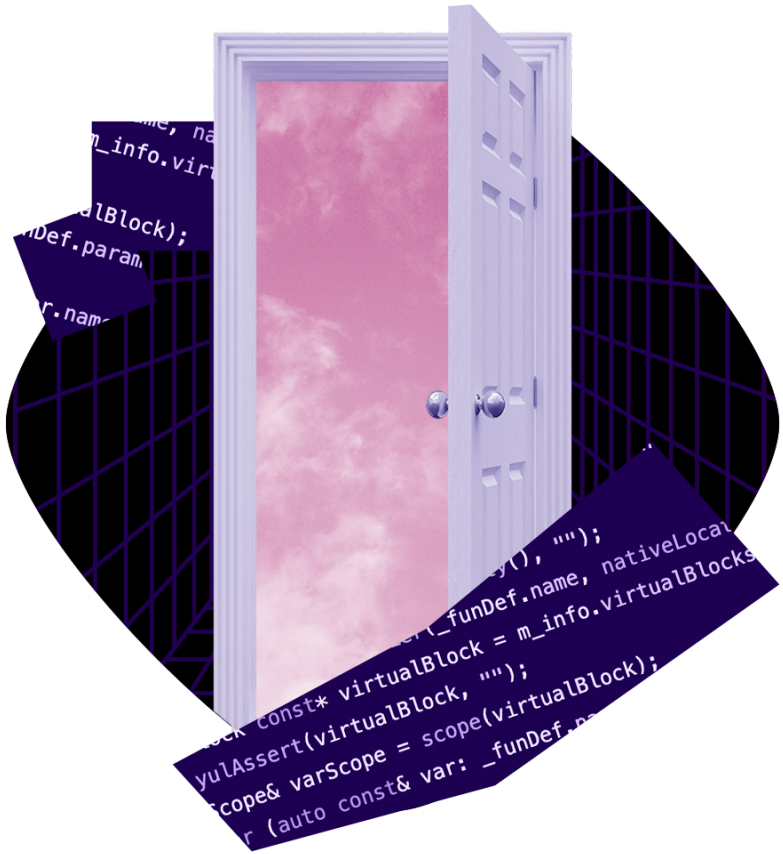
1
second block time

1
single-slot finality

MonadBFT
consensus

Decentralization Meets Scale

Monad supports 10,000 transactions per second, significantly increasing throughput capabilities and opening doors for distributed applications–even those with greater complexity and higher usage–to run in a decentralized manner.



Superscalar Architecture

Existing blockchains are extremely slow by modern computing standards. Monad is built with performance in mind, bridging the gap between decentralized and traditional platforms through superscalar, pipelined execution and optimized architecture.

Portability & Core Composability

Monad preserves full compatibility with EVM bytecode and the Ethereum RPC API–which means seamless portability for EVM developers who account for 98% of on-chain TVL across all ecosystems. Supporting all of this in a single shard allows for powerful, composable applications built on top of a global store of truth.

Join the Community

News, Press, Media
Privacy Policy
Branding & Media Kit

Our Newsletter

Drop your email to get the latest on Monad happenings, directly from the source. No fomo.

email address

SUBSCRIBE

MONAD LABS

TEAM CAREERS NEWS

The Team

Meet the people behind the tech.

Our collective experience in low-latency programming and distributed systems design runs deep, with decades under our belt building performant systems for high-frequency trading, kernel device drivers, and fintech.

Together, we are solving the unsolved and building towards a more performant future in blockchain technology.

Keone

CO-FOUNDER & CEO

in

VIEW BIO

James

CO-FOUNDER & CTO

in

VIEW BIO

Eunice

CO-FOUNDER & COO

in

VIEW BIO

Ariq

ENGINEERING

in

VIEW BIO

Aashish
ENGINEERING

in

VIEW BIO

Alex
ENGINEERING

in

VIEW BIO

Bill
COMMUNITY

VIEW BIO

Michael
ENGINEERING

in

VIEW BIO

Vicky
ENGINEERING

in

VIEW BIO

Tong
ENGINEERING

in

VIEW BIO

Kiernan
ECOSYSTEM GROWTH

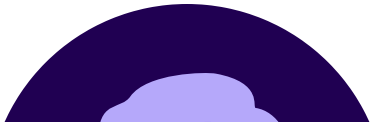
in

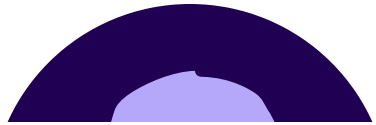
VIEW BIO

René
ENGINEERING

in

VIEW BIO



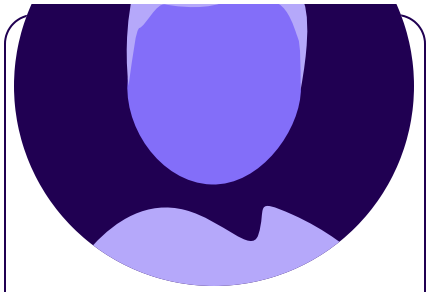


Joey

BUSINESS DEVELOPMENT

in

VIEW BIO



Niall

ENGINEERING

in

VIEW BIO

Kevin

ECOSYSTEM GROWTH

in

VIEW BIO

Abdul

BUSINESS DEVELOPMENT

in

VIEW BIO

John

ENGINEERING

Candace

RECRUITING

Kevin

DEVELOPER RELATIONS

Annie

OPERATIONS

in

VIEW BIO

in

VIEW BIO

in

VIEW BIO

in

VIEW BIO

Anna

OPERATIONS

VIEW BIO

Kai Jun

ENGINEERING

in

VIEW BIO

Bharath

ENGINEERING

in

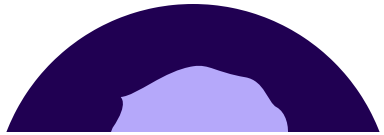
VIEW BIO

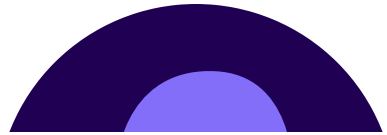
Mussadiq

RESEARCH

in

VIEW BIO





cryptunez

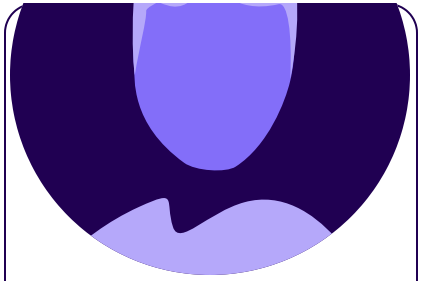
ECOSYSTEM
MARKETING

VIEW BIO

Mike

ENGINEERING

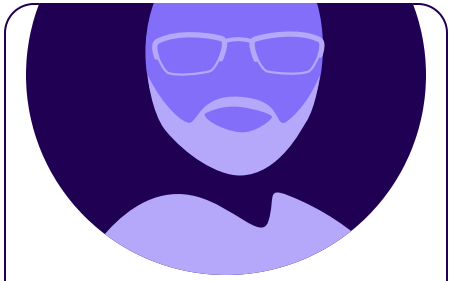
VIEW BIO



Danny

ECOSYSTEM
GROWTH

VIEW BIO



Klemens

ENGINEERING

VIEW BIO

Maged

ENGINEERING

Tina

ECOSYSTEM
INCUBATION

VIEW BIO

VIEW BIO

WHAT WE'RE BUILDING

JOIN US!

Privacy Policy

News, Press, Media

Huff resources

Huff is most closely described as EVM assembly. Unlike Yul, Huff does not provide control flow constructs or abstract away the inner working of the program stack. Only the most upmost performance sensitive applications take advantage of Huff, however it is a great educational tool to learn how the EVM interprets instructions its lowest level.

- Huff resources provides additional resources

Previous
Vyper resources

Next
Official Links

Last modified 4mo ago

Vyper resources

Vyper is a popular programming language for the EVM that is logically similar to Solidity and syntactically similar with Python.

The Vyper documentation covers installing the Vyper language, language syntax, coding examples, compilation.

A typical EVM developer looking for a Python-like experience is encouraged to use Vyper as the programming language and ApeWorx, which leverages the Python language, as the testing and deployment framework. ApeWorx also allows for the use of typical Python libraries in analysis of testing results such as Pandas.

Vyper and ApeWorx can be used with Jupyter, which offers an interactive environment using a web browser. A quick setup guide for working with Vyper and Jupyter for smart contract development for the EVM can be found here.

Resources

- Vyper by Example

- Snekmate: a Vyper library of gas-optimized smart contract building blocks
- Curve contracts: the most prominent example usage of Vyper

Previous
Other languages

Next
Huff resources

Last modified 4mo ago

Debugging on-chain

Transaction introspection/tracing

- Tenderly
- EthTx Transaction Decoder
- https://openchain.xyz/
- Bloxy
- https://github.com/naddison36/tx2uml - OS tools for generating UML diagrams
- https://github.com/apeworx/evm-trace - tracing tools

Contract Decompilation

- https://oko.palkeo.com/: a hosted version of the Panoramix decompiler

Previous
Further Solidity res...

Next
Other languages

Last modified 4mo ago

Further Solidity resources

Here are some resources beyond the ones mentioned in Suggested Resources:

Tutorials

- Ethernaut: learn by solving puzzles

Best practices/patterns

- DeFi developer roadmap
- RareSkills Book of Gas Optimization

Testing

- Echidna: fuzz testing
- Slither: static analysis for vulnerability detection

- solidity-coverage: code coverage for Solidity testing

Smart contract archives

- Smart contract sanctuary - contracts verified on Etherscan
- EVM function signature database

Previous
EVM behavior

Next
Debugging on-chain

Last modified 4mo ago

EVM behavior

EVM behavioral specification

- Notes on the EVM: straightforward technical specification of the EVM plus some behavioral examples
- EVM: From Solidity to bytecode, memory and storage: a 90-minute talk from Peter Robinson and David Hyland-Wood
- EVM illustrated: an excellent set of diagrams for confirming your mental model
- EVM Deep Dives: The Path to Shadowy Super-Coder

Opcode reference

evm.codes: opcode reference (including gas costs) and an interactive sandbox for stepping through bytecode execution

Solidity storage layout

The EVM allows smart contracts to store data in 32-byte words ("storage slots"), however the details of how complex datastructures such as lists or mappings is left as an implementation detail to the higher-level language. Solidity has a specific way of assigning variables to storage slots, described below:

- Official docs on storage layout
- Storage patterns in Solidity

Previous
Suggested Resourc...

Next
Further Solidity res...

Last modified 4mo ago

Suggested Resources

Monad is fully EVM bytecode-compatible, with all supported opcodes and precompiles as of the Shanghai fork. Monad also preserves the standard Ethereum JSON-RPC interfaces.

As such, most development resources for Ethereum Mainnet apply to development on Monad.

This page suggests a **minimal** set of resources for getting started with building a decentralized app for Ethereum. Child pages provide additional detail or options.

As Solidity is the most popular language for Ethereum smart contracts, the resources on this page focus on Solidity; alternatively see resources on Vyper or Huff. Note that since smart contracts are composable, contracts originally written in one language can still make calls to contracts in another language.

IDEs

- Remix is an interactive Solidity IDE. It is the easiest and fastest way to get started coding and compiling Solidity smart

contracts without the need for additional tool installations.

- VSCode + Solidity extension

Basic Solidity

- CryptoZombies is a great end-to-end introduction to building dApps on the EVM. It provides resources and lessons for anyone from someone who has never coded before, to experienced developers in other disciplines looking to explore blockchain development.
- Solidity by Example introduces concepts progressively through simple examples; best for developers who already have basic experience with other languages.

Intermediate Solidity

- The Solidity Language official documentation is an end-to-end description of Smart Contracts and blockchain basics centered on EVM environments. In addition to Solidity Language documentation, it covers the basics of compiling your code for deployment on an EVM as well as the basic components relevant to deploying a Smart Contract on an EVM.
- Solidity Patterns repository provides a library of code templates and explanation of their usage.
- The Uniswap V2 contract is a professional yet easy to digest smart contract that provides a great overview of an in-

production Solidity dApp. A guided walkthrough of the contract can be found here.

- Cookbook.dev provides a set of interactive example template contracts with live editing, one-click deploy, and an AI chat integration to help with code questions.
- OpenZeppelin provides customizable template contract library for common Ethereum token deployments such as ERC20, ERC712, and ERC1155. Note, they are not gas optimized.

Advanced Solidity

- The Solmate repository and Solady repository provide gas-optimized contracts utilizing Solidity or Yul.
- Yul is a intermediate language for Solidity that can generally be thought of as inline assembly for the EVM. It is not quite pure assembly, providing control flow constructs and abstracting away the inner working of the stack while still exposing the raw memory backend to developers. Yul is targeted at developers needing exposure to the EVM's raw memory backend to build high performance gas optimized EVM code.
- Huff is most closely described as EVM assembly. Unlike Yul, Huff does not provide control flow constructs or abstract away the inner working of the program stack. Only the most upmost performance sensitive applications take advantage of Huff, however it is a great educational tool to learn how the EVM interprets instructions its lowest level.

Local nodes

Developers often find it helpful to be able to run a 1-node Ethereum network with modified parameters to test interaction with the blockchain:

- Anvil is a local Ethereum node packaged in the Foundry toolkit
- Hardhat Network is a local Ethereum node packaged in the Hardhat toolkit

Installation is most easily done as part of installing the respective toolkit, described in the next section.

Toolkits

Developers often find it helpful to build their project in the context of a broader framework that organizes external dependencies (i.e. package management), organizes unit and integration tests, defines a deployment procedure (against local nodes, testnet, and mainnet), records gas costs, etc.

Here are the two most popular toolkits for Solidity development:

- Foundry is a Solidity framework for both development and testing. Foundry manages your dependencies, compiles your project, runs tests, deploys, and lets you interact with the chain from the command-line and via Solidity scripts.

Foundry users typically write their smart contracts and tests in the Solidity language.

- Hardhat is a Solidity development framework paired with a JavaScript testing framework. It allows for similar functionality as Foundry, and was the dominant toolchain for EVM developers prior to Foundry.

Interacting with the Ethereum RPC API

Frontends for dapps typically use JavaScript or Python to submit read or write queries to an RPC node. This code is typically referred to as the "client side", as web developers can roughly equate the blockchain to a backend server.

A few libraries provide standard methods for submitting queries or transactions to an RPC node:

- Python:
  - web3.py
- Javascript:
  - web3.js
  - ethers.js To this end Web3.js and Web3.py, Java Script and Python libraries respectively, have developed to make interacting with blockchains more intuitive for web developers.

Here is a quick example for creating a frontend: create-eth-app.

Testnets

Monad Testnet will be available for developers in the coming months, however, being bytecode and RPC compatible with the EVM means developers wishing to deploy on Monad can preliminary use Ethereum's Testnets.

Further details

Child pages add additional resources:

- EVM behavior
- Further Solidity resources
- Debugging on-chain
- Other languages
  - Vyper
  - Huff

Previous
Developing on Mo...

Next
EVM behavior

Last modified 4mo ago

Hardware requirements

The following hardware requirements are expected to run a Monad full node:

- CPU: 16 core CPU
- Memory: 32 GB RAM
- Storage: 2 TB NVMe SSD
- Bandwidth: 100 Mb/s

Previous
Running a node

Next
Developing on Mo...

Last modified 4mo ago

Running a node

Here are the articles in this section:

Hardware requirements

Previous
Using Monad

Next
Hardware require...

Using Monad

Here are the articles in this section:

Running a node

Developing on Monad

Previous
Other details

Next
Running a node

Other details

Accounts

Accounts in Monad are identical to Ethereum accounts. Accounts use the same address space (20-byte addresses using ECDSA). As in Ethereum, there are Externally-Owned Accounts (EOAs) and Contract Accounts.

Transactions

The transaction format in Monad matches Ethereum, i.e. it complies with EIP-2718, and transactions are encoded with RLP.

Access lists (EIP-2930) are supported but not required.

Linearity of Blocks and Transactions

Blocks are still linear, as are transactions within a block. Parallelism is utilized strictly for efficiency; it never affects the true outcome or end state of a series of transactions.

Gas

Gas (perhaps more clearly named "compute units") functions as it does in Ethereum, i.e. each opcode costs a certain amount of gas. Gas costs per opcode are identical to Ethereum in Monad, although this is subject to change in the future.

When a user submits a transaction, they include a gas limit (a max number of units of gas that this function call can consume before erroring) as well as a gas price (a bid, in units of native token, per unit of gas).

Leaders in the default Monad client use a priority gas auction (PGA) to order transactions, i.e. they order transactions in descending gas price order. In future times there may be alternative mechanisms for transaction ordering. The choice of order is orthogonal to everything that happens downstream; valid choice of order is not enshrined into the Monad protocol.

Previous
Transaction lifecycl...

Next
Using Monad

Last modified 4mo ago

Transaction lifecycle in Monad

Transaction submission

The lifecycle of a transaction starts with a user preparing a signed transaction and submitting it to an RPC node.

Transactions are typically prepared by an application frontend, then presented to the user's wallet for signing. Most wallets make an `eth_estimateGas` RPC call to populate the gas **limit** for this transaction, although the user can also override this in their wallet. The user is also typically asked to choose a gas **price** for the transaction, which is a number of NativeTokens per unit of gas.

After the user approves the signing in their wallet, the signed transaction is submitted to an RPC node using the `eth_sendTransaction` or `eth_sendRawTransaction` API call.

Of note, in Monad, the gas limit is the sum of the carriage gas and the execution gas. Carriage gas is a constant.

Mempool propagation

The RPC node forwards the pending transaction to other Monad nodes that are participating in consensus. The set of pending transactions is colloquially referred to as the 'mempool'. See Mempool for additional details on mempool behavior.

For spam prevention reasons, nodes add the transaction to their mempool only if there is sufficient gas in the reserve balance.

Block inclusion

MonadBFT uses a rotating leader mechanism to produce blocks. Each round, a leader assembles a block from the list of pending transactions. After choosing a transaction to be included in the block, the leader decrements the reserve balance with the carriage cost.

Blocks are propagated through the network as discussed in MonadBFT. The quorum certificate (QC) for this block is propagated in the subsequent round of consensus (i.e. is sent out by the next leader). After seeing the QC, voting nodes send each other votes; when a node sees 2/3 of the stake weight vote yes, it finalizes that block.

Once the block is finalized, the transaction has officially "occurred" in the history of the blockchain. Since its order is

determined, its truth value (i.e., whether it succeeds or fails, and what the outcome is immediately after that execution) is determined.

Local execution

As soon as a node finalizes a block, it begins executing the transactions from that block. For efficiency reasons, transactions are executed optimistically in parallel, but it is as if the transactions were executed serially, since results are always committed in the original order.

Querying the outcome

The user can query the result of the transaction by calling `eth_getTransactionByHash` or `eth_getTransactionReceipt` on any RPC node. The RPC node will return as soon as execution completes locally on the node.

Previous
MonadDb

Next
Other details

Last modified 5mo ago

MonadDb

MonadDb is a custom database for storing blockchain state.

Most Ethereum clients use key-value databases that are implemented as either B-Tree (an example is LMDB) or LSM-Tree (examples are LevelDB and RocksDB) data structures. However Ethereum uses the Merkle Patricia Trie (MPT) data structure for storing state. This results in a suboptimal solution where one data structure is embedded into another data structure of a different type. MonadDb implements a Patricia Trie data structure natively, both on-disk and in-memory.

Monad executes multiple transactions in parallel. When one transaction needs to read state from disk, one should not block waiting for that operation to complete - instead one should initiate the read and then start working on another transaction in the meantime. Therefore the problem needs asynchronous i/o (async i/o) for the database. The above mentioned key-value databases lack proper async i/o support (although there are some efforts to improve in this area). MonadDb fully utilizes the latest kernel support for async i/o (on Linux this is io_uring). This avoids needing to spawn a large number of kernel threads to

handle pending i/o requests in an attempt to perform work asynchronously.

MonadDb makes a number of other optimizations related to i/o, such as bypassing the filesystem which add expensive overhead.

Previous
Parallel Execution

Next
Transaction lifecycl...

Last modified 4mo ago

Parallel Execution

Monad executes transactions in parallel. While at first it might seem like this implies different execution semantics than exist in Ethereum, it actually does not. Monad blocks are the same as Ethereum blocks - a linearly ordered set of transactions. The result of executing the transactions in a block is identical between Monad and Ethereum.

Optimistic Execution

At a base level, Monad uses optimistic execution. This means that Monad will start executing transactions before earlier transactions in the block have completed. Sometimes (but not always) this results in incorrect execution.

Consider two transactions (in this order in the block):

1. Transaction 1 reads and updates the balance of account A (for example, it receives a transfer from account B).
2. Transaction 2 also reads and updates the balance of account A (for example, it makes a transfer to account C).

If these transactions are run in parallel and transaction 2 starts running before transaction 1 has completed, then the balance it reads for account A may be different than if they were run sequentially. This could result in incorrect execution.
The way optimistic execution solves this is by tracking the inputs used while executing transaction 2 and comparing them to the outputs of transaction 1. If they differ, we have detected that transaction 2 used incorrect data while executing and it needs to be executed again with the correct data.

While Monad executes transactions in parallel, the updated state for each transaction is "merged" sequentially in order to check the condition mentioned above.

Related computer science topics are optimistic concurrency control (OCC) and software transactional memory (STM).

Optimistic Execution Implications

In a naïve implementation of optimistic execution, one doesn't detect that a transaction needs to be executed again until earlier transactions in the block have completed. At that time, the state updates for all the earlier transactions have been merged so it's not possible for the transaction to fail due to optimistic execution a second time.

There are steps in executing a transaction that do not depend on state. An example is signature recovery, which is an expensive computation. This work does not need to be repeated when executing the transaction again.

Furthermore, when executing a transaction again due to failure to merge, often the account(s) and storage accessed will not change. This state is still be cached in memory, so again this is expensive work that does not need to be repeated.

Scheduling

A naïve implementation of optimistic execution will try to start executing the next transaction when the processor has available resources. There may be long "chains" of transactions which depend on each other in the block. Executing these transactions in parallel would result in a significant number of failures.

Determining dependencies between transactions ahead of time allows Monad to avoid this wasted effort by only scheduling transactions for execution when prerequisite transactions have completed. Monad has a static code analyzer that tries to make such predictions. In a good case Monad can predict many dependencies ahead of time; in the worst case Monad falls back to the naïve implementation.

Further Work

There are other opportunities to avoid re-executing transactions which are still being explored.

Previous
Execution

Next
MonadDb

Last modified 4mo ago

MONAD

Search ^ K

Powered By **GitBook**

Execution

Monad's execution improvements come in a few key areas:

- Optimistic parallel execution: parallel execution of transactions while committing results in the original order
- MonadDb: high-performance state storage backend

Previous
Carriage Cost and ...

Next
Parallel Execution

Last modified 5mo ago

Carriage Cost and Reserve Balance

Motivation

Deferring execution is incredibly powerful, as it allows execution and consensus to occur in parallel, massively expanding the time budget for execution.

An obvious objection is: now that consensus nodes don't have an up-to-date view of state, what prevents them from mistakenly including transactions from accounts that have spent all of their gas? This would create a Denial-of-Service vector.

To defend against this, Monad introduces a cost for a transaction to be carried over the network (the "carriage cost"), maintains a reserve balance for each account which is updated at consensus time, and charges carriage costs against the reserve balance.

Carriage cost

In Monad, there is a charge for having a transaction carried over the network in a block (the "carriage cost"). This is a separate charge from the cost of execution.

The carriage cost is necessary for spam prevention; the cost is minimal, but reflects the cost of utilizing network resources.

It is possible for a transaction to be included in consensus (and charged for carriage) but to have insufficient execution budget relative to the specified gas limit. In that case, at execution time, the transaction will fail but be charged gas up to the point of failure. Note that this is no different from Ethereum: transactions submitted with insufficient Ether in their account will use up Ether and fail. Charging gas up to the point of failure is necessary to prevent execution DOS vectors.

Reserve balance

For each address, nodes maintain two balances:

- a **reserve balance**, used to pay for the carriage cost

The carriage cost is charged to the reserve balance when the transaction is included in a block (consensus); it is deducted from the execution balance at execution time (double charge), and repaid to the reserve balance after the delay period of `D` blocks (10 seconds) passes.

The reserve is effectively a budget for "in-flight" orders; it exists to ensure that only transactions that are paid for are included in

blocks. You can think of the reserve balance as decrementing in realtime (i.e. as consensus occurs); although the node's view of the full state is lagged, the reserve balance always reflects up-to-date expenditures.

Target reserve balance

The target reserve balance is a per-account parameter. The default is anticipated to be a large multiple (200x) of the carriage cost, so that users can submit a large number of inflight orders without issue.

Users who anticipate sending a large number of inflight orders from the same EOA can alter the target reserve balance by interacting with an enshrined smart contract. Reserve balance alterations are treated as executions, i.e. they are only reflected in the reserve balance after the delay period passes.

Previous
Deferred Execution

Next
Execution

Last modified 4mo ago

Deferred Execution

Pipelined consensus-execution staging

One of the novel aspects of the Monad blockchain is that execution is decoupled from consensus.

To recap, consensus is the process where Monad nodes come to agreement about the official ordering of transactions, while execution is the process of actually executing those transactions and updating the state.

In Monad consensus, nodes come to agreement about the official ordering of transactions, but without either the leader or validating nodes having to have executed those transactions yet.

That is, the leader proposes an ordering without yet knowing the resultant state root, and the validating nodes vote on block validity without knowing (for example) whether all of the transactions in the block execute without reverting.

How can this be? And why does Monad do this?

The answer is a cornerstone of Monad's design, and it allows Monad to unlock significant speedups that allow a single-shard blockchain to scale to millions of users.

Interleaving execution and consensus is inefficient

In Ethereum, execution is a prerequisite to consensus, so when nodes come to consensus on a block, they are agreeing on both (1) the list of transactions in the block, and (2) the merkle root summarizing all of the state after executing that list of transactions. As a result, the leader must execute all of the transactions in the proposed block *before* sharing the proposal, and the validating nodes must execute those transactions *before* responding with a vote.

In this paradigm, the time budget for execution is extremely limited, since it has to happen twice *and* leave enough time for multiple rounds of cross-globe communication for consensus. Additionally, since execution will block consensus, the gas limit must be chosen extremely conservative, ensuring that the computation completes on all nodes within the budget even in the worst-case scenario.

Determined ordering implies state determinism

Here is an obvious but crucial insight: given an official ordering of transactions, the true state is completely determined. Execution is required to unveil the truth, but the truth is already determined.

Monad takes advantage of this insight by removing the requirement that nodes execute prior to consensus. Node agreement is purely about the official ordering; each node executes the transactions in block `N` independently while commencing consensus on block `N+1`. This allows for a gas budget corresponding to the full block time, since execution merely needs to keep up with consensus. Additionally, this approach is more tolerant of variation in exact compute time, since execution only needs to keep up with consensus on average.

Delayed merkle roots still ensure state machine replication

The main objections one might have to the above are:

- What happens if one of the nodes is malicious, so doesn't execute the exact transactions specified in consensus? (For example, say it omits certain transactions, or just sets a state variable to an arbitrary value of its choice.)
- What happens if one of the nodes makes a mistake in execution?

To address these concerns, in Monad the block proposals include a merkle root delayed by `D` blocks, where `D` is a systemwide parameter (currently anticipated to be 10). As a result of this delayed merkle root:

1. After the network comes to consensus (2/3 majority vote) on block `N`, it means that the network has agreed that the

official consequence of block `N-D` is a state rooted in merkle root `M`. Light clients can then query full nodes for merkle proofs of state variable values at block `N-D`.
2. Any node with an error in execution at block `N-D` will fall out of consensus starting at block `N`. This will trigger a rollback on that node to the end state of block `N-D-1`, followed by re-execution of the transactions in block `N-D` (hopefully resulting in the merkle root matching), followed by a re-execution of the transactions in block `N-D+1`, `N-D+2`, etc.

know that the supermajority agrees about the official ordering and the state resulting from that ordering. However, this strictness comes at great cost - extremely limited throughput. Monad loosens the strictness slightly, to great effect.

On finality

In MonadBFT, finality is single-slot (1 second), and execution outcome will generally lag by less than 1 second for those using a full node. Let's unpack this a bit.

Finality in Monad is single-slot (1 second). If you submit a transaction, you will see the transaction's official order (among all other transactions) after a single block. There is no potential for reordering, barring malicious action from a supermajority of the network. This makes Monad's finality significantly faster than Ethereum's (2 epochs, aka 12.8 minutes).

The execution outcome of a transaction (did it succeed or fail? what are the balances afterward?) **will generally lag finality by less than 1 second on full nodes.** Anyone who needs to know the outcome of a transaction quickly (for example, a high-frequency trader wanting to know the status of an order) can run a full node. Monad is designed to minimize the cost of full nodes; see Hardware Requirements for further information.

Anyone who wants to securely query the outcome of a transaction without running a full node can run a light client while querying a full node for balances with merkle proofs. In this case, queries will be lagged by the merkle root delay (`D=10` blocks, i.e. 10 seconds). Note that most users currently view the state of the blockchain using software (browser) wallets or via a block explorer. Neither of these usage patterns involve a light client.

Some readers might mistakenly conflate the merkle root delay (`D=10` blocks) with finality and mistakenly assume that finality is 10 blocks. That's not true. The official transaction order is determined after 1 block, after which there will be no reorgs without Byzantine behavior from a supermajority.

Previous
Shared Mempool

Next
Carriage Cost and ...

Last modified 4mo ago

MONAD

Search ^ K

Concepts

Pipelining

Asynchronous I/O

Why blockchain?

Why Monad: decentralization + performance

Consensus

MonadBFT

Shared Mempool

Deferred Execution

Shared Mempool

Mempool

Pending user transactions are stored in the mempool of each

tree for efficiency.

Transaction hashing

MonadBFT is an efficient means of coming to agreement about an arbitrary payload. However, block propagation is still a significant bottleneck; for example, a block with 10,000 transactions with 500 byte transactions will be 5 MB; blocks of this size would put undue bandwidth requirements on validator nodes.

To alleviate this issue, block proposals refer to transactions by hash only - a significant savings, as hashes are 32 bytes. Because of this, all validator mempools need to have the transactions in their own mempool when voting on proposals and committing

blocks. Transactions that are submitted to a validator's mempool are shared with other validator mempools by erasure-coding the transaction and then communicating over a broadcast tree for efficiency.

Previous
MonadBFT

Next
Deferred Execution

Last modified 4mo ago

MonadBFT

Pipelined two-phase HotStuff Consensus

MonadBFT is a high-performance consensus mechanism for achieving agreement about the transaction ordering under partially synchronous conditions in the presence of Byzantine actors. It is a derivative of HotStuff with the improvement proposed in Jolteon/DiemBFT/Fast-HotStuff which is the reduction from three rounds to two rounds by utilizing quadratic communication complexity in the event of leader timeout.

MonadBFT is a pipelined two-phase BFT algorithm with optimistic responsiveness, and with linear communication overhead in the common case and quadratic communication in the case of a timeout. As in most BFT algorithms, communication proceeds in phases; at each phase, the leader sends a signed message to the voters, who send signed responses to the following leader. Pipelining allows the quorum certificate (QC) or timeout certificate (TC) for block `k` to piggyback on the proposal for block `k+1`.

Quick facts

Sybil resistance mechanism	Proof-of-Stake (PoS)
Block time	1 second
Finality	Single-slot
Delegation allowed	Yes

Mempool

See Shared Mempool.

Consensus Protocol

MonadBFT is a pipelined consensus mechanism that proceeds in rounds. The below description gives a high-level intuitive understanding of the protocol.

As is customary, let there be `n = 3f+1` nodes, where `f` is the max number of Byzantine nodes, i.e. `2f+1` (i.e. 2/3) of the nodes are non-Byzantine. In the discussion below, let us also treat all nodes as having equal stake weight; in practice all thresholds can be expressed in terms of stake weight rather than in node count.

- In each round, the leader sends out a new block as well as either a QC or a TC (more on this shortly) for the previous round.
- Each validator reviews that block for adherence to protocol and, if they agree, send signed YES votes to the leader of the

next round. That leader derives a QC (quorum certificate) by aggregating (via threshold signatures) YES votes from `2f+1` validators. Note that communication in this case is *linear*: leader sends block to validators, validators send votes directly to next leader.

- Alternatively, if the validator does not receive a valid block within a pre-specified amount of time, it multicasts a signed timeout message to *all* of its peers. This timeout message also includes the highest QC that the validator has observed. If any validator accumulates `2f+1` timeout messages, it assembles these (again via threshold signatures) into a TC (timeout certificate) which it then sends directly to the next leader.

- Each validator finalizes the block proposed in round `k` upon receiving a QC for round `k+1` (i.e. in the communication from the leader of round `k+2`). Specifically:
  - Alice, the leader of round `k`, sends a new block to everyone (along with either a QC or TC for round `k-1`; let's ignore that as it's not important).
  - If `2f+1` validators vote YES on that block by sending their votes to Bob (leader of round `k+1`), then the block in `k+1` will include a QC for round `k`. However, seeing the QC for round `k` *at this point* is not enough for Valerie the validator to know that the block in round `k` has been enshrined, since (for example) Bob could have been malicious and only sent the block to Valerie. All that Valerie can do is vote on block `k+1`, sending her votes to Charlie (leader of round `k+2`).
  - If `2f+1` validators vote YES on block `k+1`, then Charlie publishes a QC for round `k+1` as well as a block proposal

for round `k+2`. As soon as Valerie receives this block, she knows that the block from round `k` (Alice's block) is finalized.

- Say that Bob had acted maliciously, either by sending an invalid block proposal at round `k+1`, or by sending it to fewer than `2f+1` validators. Then at least `f+1` validators will timeout, then triggering the other non-Byzantine validators to timeout, leading to at least one of

QC will be available for round `k+1`).

- We refer to this commitment procedure as a 2-chain commit rule, because as soon as a validator sees 2 adjacent certified blocks `B` and `B'`, they can commit `B` and all of its ancestors.

References:

- Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain, 2018.
- Mohammad M. Jalalzai, Jianyu Niu, Chen Feng, Fangyu Gai. Fast-HotStuff: A Fast and Resilient HotStuff Protocol, 2020.
- Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. arXiv preprint arXiv:2106.10362, 2021.

- The Diem Team. DiemBFT v4: State machine replication in the diem blockchain. 2021.

BLS Multi-Signatures

Certificates (QCs and TCs) can be naively implemented as a vector of ECDSA signatures on the secp256k1 curve. These certificates are explicit and easy to construct and verify. However, the size of the certificate is linear with the number of signers. It poses a limit to scaling because the certificate is included in almost every consensus message, except vote message.

Pairing-based BLS signature on the BLS12-381 curve helps with solving the scaling issue. The signatures can be incrementally aggregated into one signature. Verifying the single valid aggregated signature provides proof that the stakes associated with the public keys have all signed on the message.

BLS signature is much slower than ECDSA signature. So for performance reasons, MonadBFT adopts a blended signature scheme where BLS signature is only used on aggregatable message types (votes and timeouts). Message integrity and authenticity is still provided by ECDSA signatures.

Previous
Consensus

Next
Shared Mempool

Last modified 4mo ago

MONAD

Search ^ K

Consensus

Understanding Monad's consensus requires understanding of a few key areas:

- MonadBFT: Monad's consensus mechanism for achieving agreement about an arbitrary payload under partially synchronous conditions while maintaining Byzantine fault tolerance.
- Shared Mempool: Defining a significant optimization to the consensus payload: referring to transactions by hash, and ensuring that transactions are propagated through the mempool ahead of time.
- Deferred Execution: Defining a significant optimization to the process of coming to consensus, which is moving execution out of the hot path of consensus.
- Carriage Cost and Reserve Balance: Defining a behavioral change to transaction pricing which is required to defend against spam given that consensus is done over a delayed view of execution.

Previous
Why Monad: decen...

Next
MonadBFT

Last modified 5mo ago

Why Monad: decentralization + performance

Decentralization matters

A blockchain has several major components:

- Consensus mechanism for achieving agreement on transactions to append to the ledger
- Execution/storage system for maintaining the active state

In increasing the performance of these components, one could cut corners, for example by requiring all of the nodes to be physically close to each other (to save on the overhead of consensus), or by requiring a huge amount of RAM (to keep much or all of the state in memory), but it would be at a significant cost to decentralization.

And decentralization is the whole point!

As discussed in Why Blockchain?, decentralized shared global state allows many parties to coordinate while relying on a single, shared, objective source of truth. Decentralization is key to the matter; a blockchain maintained by a small group of node operators (or in the extreme case, a single operator!) would not offer benefits such as trustlessness, credible neutrality, and censorship-resistance.

For any blockchain network, decentralization should be the principal concern. Performance improvements should not come at the expense of decentralization.

Today's performance bottlenecks

Ethereum's current execution limits (1.25M gas per second) are set conservatively, but for several good reasons:

- Inefficient storage access patterns
- Single-threaded execution

- Concerns about state growth, and the effect of state growth on future state access costs

Monad addresses these limitations through algorithmic improvements and architectural changes, pioneering several innovations that will hopefully become standard in Ethereum in the years to come. Maintaining a high degree of decentralization,

while making material performance improvements, is the key consideration.

Addressing these bottlenecks through optimization

Monad enables pipelining and other optimizations in four major areas to enable exceptional Ethereum Virtual Machine performance and materially advance the decentralization/scalability tradeoff. Subsequent pages describe these major areas of improvement:

- MonadBFT
- Deferred Execution
- Parallel Execution
- MonadDb

Previous
Why blockchain?

Next
Consensus

Last modified 4mo ago

Why blockchain?

A simple mental model for the 'what' and 'why'.

A blockchain is decentralized agreement among a diverse set of participants about two things:

1. An official ordering (ledger) of transactions
2. An official state of the world, including balances of accounts and the state of various programs.

In modern blockchains such as Ethereum, transactions consist of balance transfers, creation of new programs, and function calls against existing programs. The aggregate result of all transactions up to now produces the current state, which is why *agreement about (1) implies agreement about (2).*

A blockchain system has a set of protocol rules which describe how a distributed set of nodes which are currently in sync will communicate with each other to agree upon a new list of transactions that each should apply. Induction keeps the nodes in sync: they start with the same state and apply the same transactions, so at the end of applying a new list of transactions, they still have consistent state. (This essay will ignore the details

of how such a system of nodes achieves agreement, but you can refer to the documentation of Monad's consensus mechanism for further details.)

Shared global state enables the development of decentralized apps - apps that live "on the blockchain", i.e. on each of the nodes in the blockchain system. A decentralized app is a chunk of code (as well as persistent, app-specific state) that can get invoked by any user, who does so by submitting a transaction pointing to a function on that app. Each of the nodes in the blockchain is responsible for correctly executing the bytecode being called; duplication keeps each node honest.

An example app

Decentralized apps can implement functionality that we might otherwise expect to be implemented in a centralized fashion. For example, a very simple example of a decentralized app is a *Virtual Bank* (typically referred to in crypto as a Lending Protocol).

In the physical world, a bank is a business that takes deposits and loans them out at a higher rate. The bank makes the spread between the high rate and the low rate; the borrower gets a loan to do something economically productive; and you earn interest on your deposits. Everyone wins!

A Virtual Bank is simply an app with four major methods: `deposit`, `withdraw`, `borrow`, and `repay`. The logic for each of those methods is mostly bookkeeping to ensure that deposits and loans are being tracked correctly:

```
class VirtualBank:
```

```
def deposit(sender, amount):
  # transfer amount from sender to myself (the bank)
  # do internal bookkeeping to credit the sender

def withdraw(sender, amount):
  # ensure the sender had enough on deposit
  # do internal bookkeeping to debit the sender
  # transfer amount from myself (the bank) to sender
```



```
def repay(sender, amount);
  # ...
```

In Ethereum, or in Monad, someone can write code for this Virtual Bank and upload it; then anyone can utilize it for borrowing and lending, potentially far more easily than when trying to get access to banking services in their home country.

This simple example shows the power of decentralized apps. Here are a few other benefits to call out:

- **Open APIs / composability**: decentralized apps can be called atomically by other decentralized apps, allowing developers to build more complex functionality by stacking existing components.
- **Transparency**: app logic is expressed purely through code, so anyone can review the logic for side effects. State is transparent and auditable; proof of reserves in DeFi is the default.

Shared Mempool

Deferred Execution

Powered By **GitBook**

- **Censorship-resistance and credible neutrality:** anyone can submit transactions or upload applications to a permissionless network.
- **Global reach**: anyone with access to the internet can access crucial financial services, including unbanked/underbanked users.

Previous
Asynchronous I/O

Next
Why Monad: decen...

Last modified 4mo ago

Asynchronous I/O

Asynchronous I/O is a form of input/output processing that allows the CPU to continue executing concurrently while communication is in progress.

Disk and network are orders of magnitude slower than the CPU. Rather than initiating an I/O operation and waiting for the result, the CPU can initiate the I/O operation as soon as it's known that the data will be needed, and continue executing other instructions which do not depend on the result of the I/O operation.

Some rough comparisons for illustration purposes:

Device	Latency	Ba
CPU L3 Cache	10 ns	>4
Memory	100 ns	10
Disk (Nvme SSD)	400 us	38
Network	50 - 200 ms	1 C

(actual disk stats as reported by fio for random reads of size 2KB - ~190k IOPS)

Fortunately, SSD drives can perform operations concurrently, so the CPU can initiate several requests at the same time, continue executing and then receive the results of multiple operations around the same time.

Some databases (such as lmdb / mdbx) use memory-mapped storage to read and write to disk. Unfortunately, memory-mapped storage is implemented by the kernel (mmap) and is not asynchronous, so execution is blocked while waiting for the operation to complete.

More about asynchronous i/o can be read here.

Previous
Pipelining

Next
Why blockchain?

Last modified 4mo ago

MONAD

Search ^ K

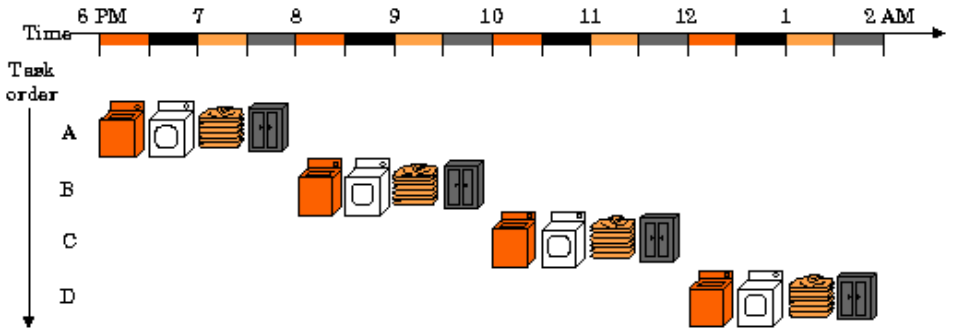
Powered By **GitBook**

Pipelining

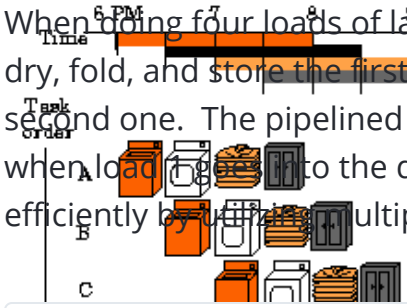
Pipelining is a technique for implementing parallelism by dividing tasks into a series of smaller tasks which can be processed in parallel.

Pipelining is used in computer processors to increase the throughput of executing a series of instructions sequentially at the same clock rate. (There are other techniques used in processors to increase throughput as well.) More about instruction-level parallelism (ILP) can be read here.

A simple example of pipelining:



When doing four loads of laundry, the naive strategy is to wash, dry, fold, and store the first load of laundry before starting on the second one. The pipelined strategy is to start washing load 2 when load 1 goes into the dryer. Pipelining gets work done more efficiently by utilizing multiple resources simultaneously.



Previous
Concepts

Next
Asynchronous I/O

Last modified 4mo ago

MONAD

Search ^ K

Concepts

Here are the articles in this section:

Pipelining

Asynchronous I/O

Previous
Technical discussion

Next
Pipelining

MONAD

Search ^ K

Technical discussion

The technical discussion is divided into several major areas:

- Concepts: reviewing a few crucial concepts from Computer Science
- Why blockchain?: offering a mental model for the value of decentralized shared global state
- Why Monad: decentralization + performance: summarizing some of the existing bottlenecks in maintaining shared global state in Ethereum, and how Monad addresses them
- Consensus: a summary of the novel aspects of Monad's mempool and consensus layers
- Execution: a summary of how transactions are executed in Monad, as well as how state is stored
- Transaction lifecycle: a walkthrough of the lifecycle of a transaction

Monad enables pipelining and optimization in four major areas to enable exceptional Ethereum Virtual Machine performance

and materially advance the decentralization/scalability tradeoff. If you'd like to focus on those areas, please see the relevant pages below:

- MonadBFT: pipelined HotStuff consensus with additional research improvements
- Deferred Execution: consensus-execution staging
- Parallel Execution
- MonadDb: high-performance state backend for merkle trie storage

Previous
Monad for develop...

Next
Concepts

Last modified 4mo ago

Monad for developers

Monad is an Ethereum-compatible Layer-1 blockchain with 10,000 tps of throughput, 1-second block times, and single-slot finality.

Monad's implementation of the Ethereum Virtual Machine complies with the Shanghai fork; simulation of historical Ethereum transactions with the Monad execution environment produces identical outcomes. Monad also offers full Ethereum RPC compatibility so that users can interact with Monad using familiar tools like MetaMask and Etherscan.

Monad accomplishes these performance improvements through the introduction of several major innovations:

- MonadBFT (pipelined HotStuff consensus with additional research improvements),
- Deferred Execution (pipelining between consensus and execution to significantly increase the execution budget)
- Parallel Execution
- MonadDb (high-performance state backend)

Although Monad features parallel execution and pipelining, it's important to note that blocks in Monad are linear, and transactions are linearly ordered within each block.

Transaction format

Address space	matches Ethereum 20-byte addresses using ECDS
Transaction format	matches Ethereum complies with EIP-2718, encoc lists (EIP-2930) are supported
Wallet compatibility	Monad is compatible with star such as Metamask. The only c alter the RPC URL and ChainId

Smart contracts

- Monad supports EVM bytecode, and is bytecode-equivalent to Ethereum. All opcodes (as of the Shanghai fork) are supported.

Consensus

Sybil resistance mechanism	Proof-of-Stake (PoS)
Delegation	Allowed (in-protocol)
Consensus mechanism and pipelining	MonadBFT is a leader-based algorithn agreement about transaction order ar partially synchronous conditions. Broa a derivative of HotStuff with additiona improvements. MonadBFT is a pipelined 2-phase BFT communication overhead in the comr BFT algorithms, communication proce phase, the leader sends a signed mes send back signed responses. Pipelinir certificate (QC) or timeout certificate (piggyback on the proposal for block k quadratic messaging.
Block time	1 second
Finality	Single-slot. Once 2/3 of the stake weig block proposal, it is finalized.
Mempool	There is a mempool. Transactions are communicated using a broadcast tree
Spam resistance	Users pay for inclusion in blocks ("carı transaction execution ("execution cos

Consensus participants	Direct consensus participants vote on serve as leaders. To serve as a direct p must have at least `MinStake` staked `MaxConsensusNodes` participants by parameters are set in code.
Transaction hashing	For efficiency, block proposals refer to only. If a node does not have a transa transaction by hash from a neighbor.
Deferred execution and carriage costs	In Monad, consensus and execution o fashion. Nodes come to consensus or transaction order *prior* to executing th Execution); the outcome of execution consensus. In blockchains where execution *is* a pr consensus, the time budget for execu of the block time. Pipelining consensu Monad to expend the full block time c execution. Block proposals consist of an ordered hashes and a state merkle root from delay parameter `D` is set in code; it is initially. The user must pay to have a transacti (the "carriage cost"). For each addres balances:

	• a reserve balance, used to pay for • the execution balance, used to pa execution Carriage cost is charged to the reserve transaction is included in a block (con from the execution balance at executi charge), and repaid to the reserve bal period of `D` blocks passes. An account's reserve balance is effect flight" orders; it exists to ensure that c are paid for are included in blocks. Each account has a target reserve bal altered by interacting with an enshrin for EOAs that anticipate sending a larg orders.
State determinism	Finality occurs at consensus time; the transactions is enshrined at this point fully deterministic for any full node, w execute the transactions for that new second.

Execution

The execution phase for each block begins after consensus is reached on that block, allowing the node to proceed with consensus on subsequent blocks.

Parallel Execution

Transactions are linearly ordered; the job of execution is to arrive at the state that results from executing that list of transactions serially. The naive approach is just to execute the transactions one after another. Can we do better? Yes we can!

Monad implements parallel execution:

- An executor is a virtual machine for executing transactions. Monad runs many executors in parallel.
- An executor takes a transaction and produces a **result**. A result is a list of **inputs** to and **outputs** of the transactions, where inputs are (ContractAddress, Slot, Value) tuples that were SLOADed in the course of execution, and outputs are (ContractAddress, Slot, Value) tuples that were SSTOREd as a result of the transaction.
- Results are initially produced in a pending state; they are then committed in the original order of the transactions. When a result is committed, its outputs update the current state. When it is a result's turn to be committed, Monad checks that its inputs still match the current state; if they don't, Monad reschedules the transaction. As a result of this concurrency control, Monad's execution is guaranteed to produce the same result as if transactions were run serially.
- When transactions are rescheduled, many or all of the required inputs are cached, so re-execution is generally relatively inexpensive. Note that upon re-execution, a

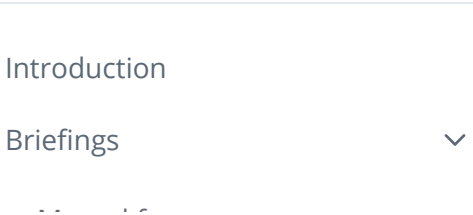
transaction may produce a different set of Inputs than the previous execution did;

MonadDb: high-performance state backend

All active state is stored in MonadDb, a storage backend for solid-state drives (SSDs) that is optimized for storing merkle trie data. Updates are batched so that the merkle root can be updated efficiently.

MonadDb implements in-memory caching and uses asio for efficient asynchronous reads and writes. Nodes should have 32 GB of RAM for optimal performance.

Comparison to Ethereum: User's Perspective



Attribute	Ethereum
Transactions/second (smart contract calls and transfers)	~10
Block time	12s
Finality	2 epochs (12-18 min)
Bytecode standard	EVM (Shanghai fork)
RPC API	Ethereum RPC API
Cryptography	ECDSA
Accounts	Last 20 bytes of keccak-256 of public key under ECDSA
Consensus mechanism	Gasper (Casper-FFG finality gadget + LMD-GHOST fork-choice rule)
Mempool	Yes
Transaction ordering	Leader's discretion (in practice, PBS)
Sybil-resistance mechanism	PoS

Previous

Next

Monad for users	Technical discussion
	No through LSTs
Hardware	4-core CPU

Last modified 3mo ago

MONAD

Search ^ K

Monad for users

Monad is a high-performance Ethereum-compatible L1, offering users the best of both worlds: **portability** and **performance**.

From a portability perspective, Monad offers **full bytecode compatibility** for the Ethereum Virtual Machine (EVM), so that applications built for Ethereum can be ported to Monad without code changes, and **full Ethereum RPC compatibility**, so that infrastructure like MetaMask or Etherscan can be used seamlessly.

From a performance perspective, Monad offers **10,000 tps** of throughput, i.e. 1 billion transactions per day, while offering **1 second block times** and **1 second finality**. This allows Monad to support many more users and far more interactive experiences than existing blockchains, while offering far cheaper per-transaction costs.

What's familiar about Monad?

From a user perspective, Monad behaves very similarly to Ethereum. You can use the same wallets (e.g. MetaMask) or block explorers (e.g. Etherscan) to sign or view transactions. The same

apps built for Ethereum can be ported to Monad without code changes, so it is expected that you'll be able to use many of your favorite apps from Ethereum on Monad. The address space in Monad is the same as in Ethereum, so you can reuse your existing keys.

Like Ethereum, Monad features linear blocks, and linear ordering of transactions within a block.

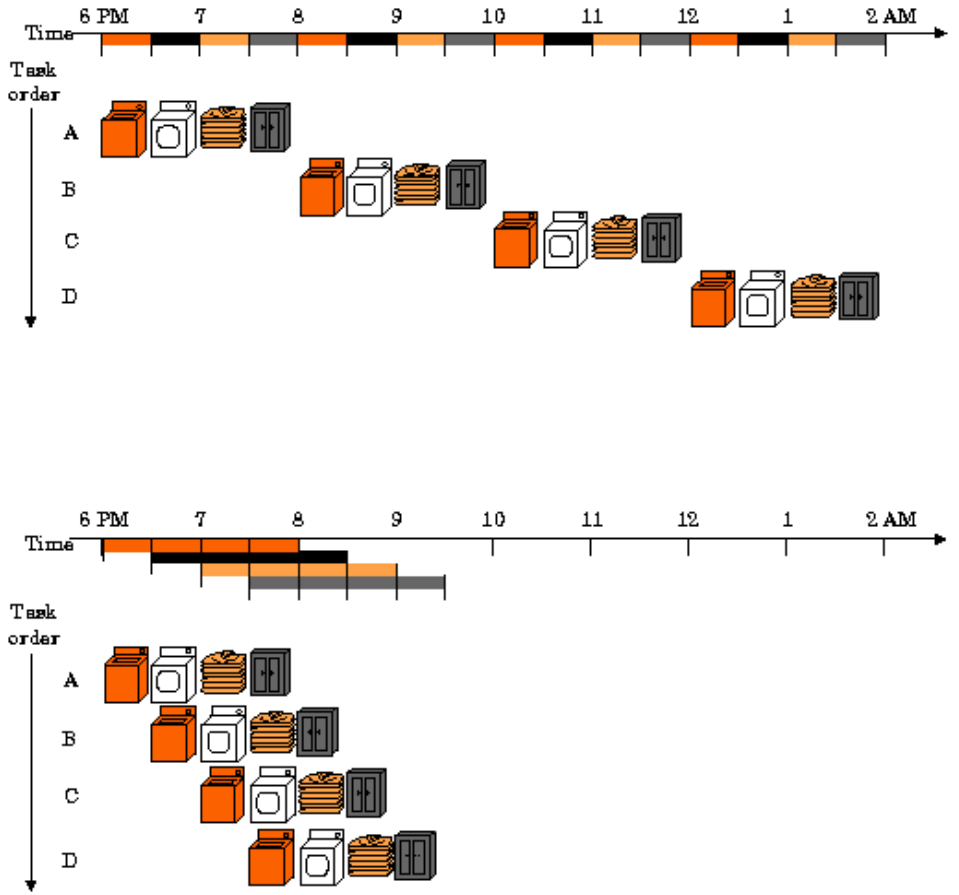
Like Ethereum, Monad is a proof-of-stake network maintained by a decentralized set of validators. Anyone can run a node to independently verify transaction execution, and significant care has been taken to keep hardware requirements minimal.

What's different about Monad?

Monad makes exceptional performance possible by introducing **parallel execution** and **superscalar pipelining** to the Ethereum Virtual Machine.

Parallel execution is the practice of utilizing multiple cores and threads to strategically execute work in parallel while still committing the results in the original order. Although transactions are executed in parallel "under the hood", from the user and developer perspective they are executed serially; the result of a series of transactions is always the same as if the transactions had been executed one after another.

Superscalar pipelining is the practice of creating stages of work and executing the stages in parallel. A simple diagram tells the story:



A familiar example of pipelining: doing laundry intelligently. Top: naive; bottom: pipelined. Credit: Prof. Lois Hawkes, FSU

When doing four loads of laundry, the naive strategy is to wash, dry, fold, and store the first load of laundry before starting on the second one. The pipelined strategy is to start washing load 2 when load 1 goes into the dryer. Pipelining gets work done more efficiently by utilizing multiple resources simultaneously.

Monad introduces pipelining to address existing bottlenecks in state storage, transaction processing, and distributed consensus.

In particular, Monad introduces pipelining and other optimizations in four major areas:

- MonadBFT (pipelined HotStuff consensus with additional research improvements)
- Deferred Execution (pipelining between consensus and execution to significantly increase the execution budget)
- Parallel Execution
- MonadDb (high-performance state backend)

Monad's client, which was written from scratch in C++ and Rust, reflect these architectural improvements and result in a platform for decentralized apps that can truly scale to world adoption.

Why should I care?

Decentralized apps are replacements for centralized services with several significant advantages:

- **Open APIs / composability**: decentralized apps can be called atomically by other decentralized apps, allowing developers to build more complex functionality by stacking existing components.
- **Transparency**: app logic is expressed purely through code, so anyone can review the logic for side effects. State is transparent and auditable; proof of reserves in DeFi is the default.

- **Censorship resistance and *credible neutrality*:** anyone can submit transactions or upload applications to a permissionless network.
- **Global reach**: anyone with access to the internet can access crucial financial services, including unbanked/underbanked users.

However, decentralized apps need cheap, performant infrastructure to reach their intended level of impact. A single app with 1 million daily active users (DAUs) and 10 transactions per user per day would require 10 million transactions per day, or 100 tps. A quick glance at EthTPS.info - a useful website summarizing the throughput of existing EVM-compatible L1s and L2s - shows that no EVM blockchain supports even that level of throughput right now.

Monad materially improves on the performance of an EVM-compatible blockchain network, pioneering several innovations that will hopefully become standard in Ethereum in the years to come.

With Monad, developers, users, and researchers can reuse the wealth of existing applications, libraries, and applied cryptography research that have all been built for the EVM.

How do I use Monad?

Monad's first public testnet will go live in the coming months.

When it does, you will be able to add an appropriate RPC url and ChainId to your Ethereum-compatible wallet and begin using

Monad like you would any other Ethereum-compatible network. Until then, stay tuned!

Previous
Briefings

Next
Monad for develop...

Last modified 4mo ago

MONAD

Search ^ K

Briefings

Here are several versions of the same briefing, depending on the user's knowledge of / interest in blockchain systems:

- Monad for users
- Monad for developers

Previous
Introduction

Next
Monad for users

Last modified 4mo ago

MONAD

Search ^ K

Introduction

Monad is a high-performance Ethereum-compatible L1. Monad materially advances the efficient frontier in the tradeoff between decentralization and scalability.

Monad introduces optimizations in four major areas, resulting in a blockchain with throughput of 10,000 transactions per second (tps):

- MonadBFT
- Deferred Execution
- Parallel Execution
- MonadDb

Monad's improvements address existing bottlenecks while preserving seamless compatibility for application developers (full EVM bytecode compatibility) and users (Ethereum RPC API compatibility). As a result, the rich landscape of Ethereum tooling and applied cryptography research can plug seamlessly into Monad while benefiting from improved throughput and scale:

- applications (any dapp built for Ethereum)

- developer tooling (e.g. Hardhat, Apeworx, Foundry)
- wallets (e.g. MetaMask)
- analytics/indexing (e.g. Etherscan, Dune)

The Monad client is built with a focus on performance and is written from scratch in C++ and Rust. The subsequent pages survey the major changes in Monad as well as the interface for users.

Next
Briefings →

Last modified 4mo ago