Background: Why is Merklization Expensive? Why use NOMT?

Background: Why is Merklization Expensive?

Addressable merkle trees usually work by grouping intermediate hashes into sets of 16 (called a `node`) and storing them in a key-value store like LMDB or RocksDB, where the key is the node hash and the value is the node itself.

To traverse the trie, implementations do something like the following:

```
struct ChildInfo {
  is_leaf: bool,
  hash: H256,
// ... some metadata omitted here}

struct Node {
  children: [Option<ChildInfo>; 16]
}

fnget(db: TrieDb, path: &[u4]) -> Option<Leaf> {
  let mut current_node: Node = db.get_root_node();
  for nibble in path {
    let child: ChildInfo = current_node[nibble]?;
    if child.is_leaf {
// Some checks omitted herereturn db.get_leaf(child.hash)
    }
    current_node = db.get_node(child.hash)
  }
}
```

As you can clearly see, this algorithm requires one random disk access for each layer of the tree that needs to be traversed. A basic property of addressable merkle tries is that their *expected depth* is logarithmic in the number of items stored, and the base of the logarithm is the width of the intermediate nodes. So if your blockchain has five billion objects in state and uses intermediate nodes with a width of 16, each state update will need to traverse to an expected depth of `log_16(5,000,000,000) ~= 8`. Putting all of this together, we can see that traversing a naive merkle trie will require roughly 8 *sequential* database queries per entry.

refs:

https://sovereign.mirror.xyz/jfx_cJ_15saejG9ZuQWjnGnG-NfahbazQH98i1J3NN8

B-Tree Implementation and State Storage

NOMT uses a specialized B-Tree variant designed specifically for blockchain state storage. Unlike traditional B-Trees, this implementation is optimized for handling high-entropy cryptographic keys, which are common in blockchain state storage. The tree structure maintains balance without excessive rebalancing operations, which is crucial when dealing with cryptographic hashes as keys.

Each node in the tree is carefully structured to align with SSD block sizes, minimizing read amplification during traversal. The implementation includes intelligent prefetching mechanisms that predict which nodes are likely to be accessed next based on access patterns common in blockchain operations.

Key operations are optimized for the specific patterns seen in blockchain state updates:

- Batch insertions are coalesced to minimize tree reorganization
- Deletions maintain tree density without excessive rebalancing
- Range scans are optimized for common blockchain queries like state enumeration

NOMT Architecture Detailed Analysis

Merkle Tree Storage Architecture

The Merkle tree implementation uses a sparse binary tree stored in an on-disk hashtable. This design is crucial for efficient proof generation and state root calculations. The storage format is specifically optimized for SSDs, with careful attention to write patterns to prevent wear leveling issues.

The Merkle tree store (Bitbox) maintains its data structures in a way that minimizes the number of disk accesses needed for proof generation. It uses intelligent caching strategies that keep frequently accessed tree nodes in memory while ensuring that memory usage remains predictable and bounded.

Critical optimizations include:

- Branch node compression to reduce storage overhead
- Intelligent node placement to minimize disk seeks during proof generation
- Batch proof generation optimizations for multiple keys
- Incremental root hash updates that avoid full tree traversal

I/O Subsystem Design

The I/O system leverages io_uring for asynchronous operations, but the real innovation is in how these operations are batched and scheduled. The system maintains separate queues for different types of operations (reads, writes, proofs) and uses sophisticated scheduling to maximize throughput.

Key I/O optimizations include:

- Write coalescing to minimize the number of actual disk operations
- Read-ahead predictions based on access patterns
- Intelligent buffer management that works with the kernel's page cache
- Direct I/O paths for critical operations that bypass unnecessary buffering

Transaction Management and Consistency

The session-based transaction system provides ACID guarantees while maintaining high performance. Each session maintains its own view of the state, allowing for consistent reads even during updates. The system uses a combination of copy-on-write and careful buffer management to maintain these guarantees without excessive memory overhead.

Write operations are handled through a sophisticated buffering system:

- Changes are first recorded in session-local buffers
- Updates are coalesced when possible
- Disk writes are scheduled to maintain consistency while maximizing throughput
- Recovery information is maintained without requiring explicit WAL

Memory Management and Resource Utilization

Memory usage is carefully controlled through a combination of techniques:

- Tiered buffer management with different eviction policies
- Predictable memory usage patterns that avoid sudden spikes
- Efficient page reuse that minimizes allocation overhead
- Smart caching strategies that adapt to access patterns

Performance Characteristics and Hardware Utilization

The system is specifically optimized for modern NVMe SSDs and takes advantage of their characteristics:

- Command queue depth is maintained at optimal levels
- Write patterns are optimized to work with SSD firmware
- Read operations are structured to minimize latency
- Garbage collection impact is minimized through careful write scheduling

Concurrency Model

The single-writer-multiple-reader design provides several advantages:

- No need for complex concurrency control in the core engine
- Predictable performance characteristics
- Simpler recovery mechanisms
- Clear consistency guarantees

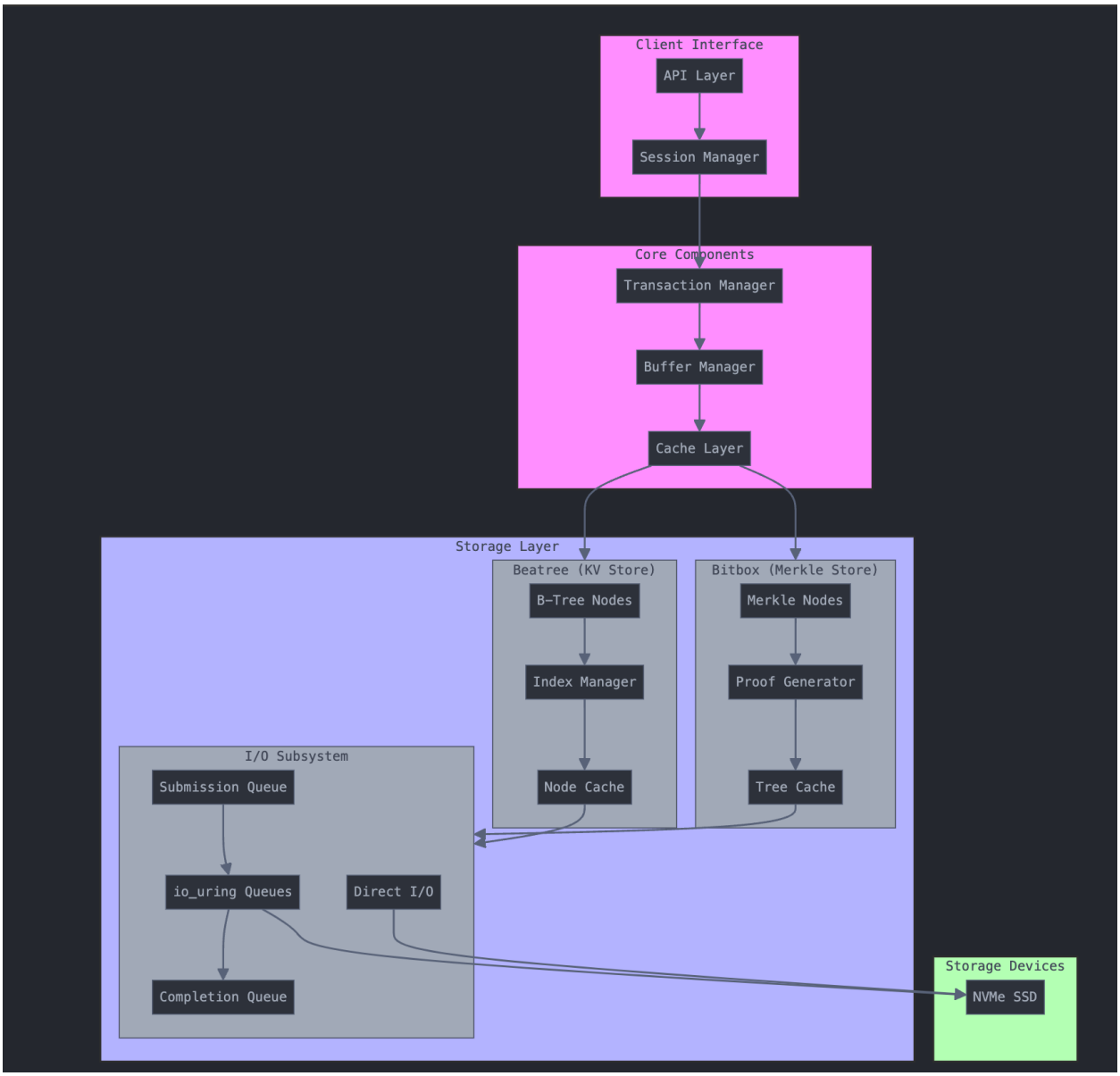
Reads are handled through a sophisticated snapshot system that provides consistent views without blocking writers.

State Management Optimizations

State updates are handled through an optimized pipeline:

- Changes are batched for efficient processing
- Merkle tree updates are minimized through clever diffing
- State root calculations are incremental
- Proof generation is optimized for common patterns

This combination of optimizations results in a system that provides significant performance advantages for blockchain state storage, while maintaining the strong consistency guarantees required for blockchain operations. The careful attention to each component's implementation and their interaction provides cumulative benefits that exceed what might be expected from any single optimization.



WHY USE NOMT IN SUBSTRATE?

Substrate currently supports two backends: RocksDB and ParityDB, so we've been mostly comparing these two in the blockchain specific workloads. ParityDb offers better read and write performance at the moment while using slightly more disk space

Substrate uses a simple key-value data store implemented as a database-backed, modified Merkle tree. All of Substrate's higher-level storage abstractions are built

on top of this simple key-value store.

Substrate implements its storage database with RocksDB, a persistent key-value store for fast storage environments. It also supports an experimental Parity DB.

The DB is used for all the components of Substrate that require persistent storage, such as:

Substrate clients
Substrate light-clients
Off-chain workers

One advantage of using a simple key-value store is that you are able to easily abstract storage structures on top of it.

Substrate uses a Base-16 Modified Merkle Patricia tree ("trie") from paritytech/trie to provide a trie structure whose contents can be modified and whose root hash is recalculated efficiently.

But the thing is substrates ParityDB primarily uses standard file I/O operations through Rust's standard library and doesn't have native io_uring support. It relies more on memory mapping (mmap) for performance.

When examining the practical advantages of NOMT for Substrate development, we need to consider how Substrate's internal architecture interacts with its storage layer. Substrate's core functionality revolves around continuous state transitions, where each block execution requires multiple state reads, writes, and Merkle proof operations. This creates a specific pattern of storage access that directly impacts blockchain performance.

The current Substrate storage implementation relies on a trie-based structure built on top of a key-value store, where every state access requires traversing this trie structure. During block execution, Substrate needs to read the existing state for transaction validation, modify state based on transaction execution, and generate new state root hashes for block finalization. These operations happen repeatedly and must be efficient to maintain good blockchain performance.
NOMT's architecture is particularly well-suited for these operations because it separates the concerns of key-value storage and Merkle tree management. When Substrate executes a transaction, it typically needs to both read the current state and prepare for state root calculations. With current implementations, these

operations can create redundant data access patterns, as the same data might need to be read multiple times for different purposes.
The session-based transaction model in NOMT aligns perfectly with Substrate's block execution model. Each block can be processed within a single session, allowing for efficient batching of operations and ensuring consistency. When Substrate processes a block, all state changes can be accumulated within a session, and the state root can be calculated efficiently without requiring separate passes over the data.

Substrate's storage overlays, which are used to manage temporary state changes during transaction execution, benefit significantly from NOMT's approach to state management. The session-based model provides a natural way to handle these temporary states, allowing for efficient validation of multiple transactions while maintaining the ability to roll back changes if needed.

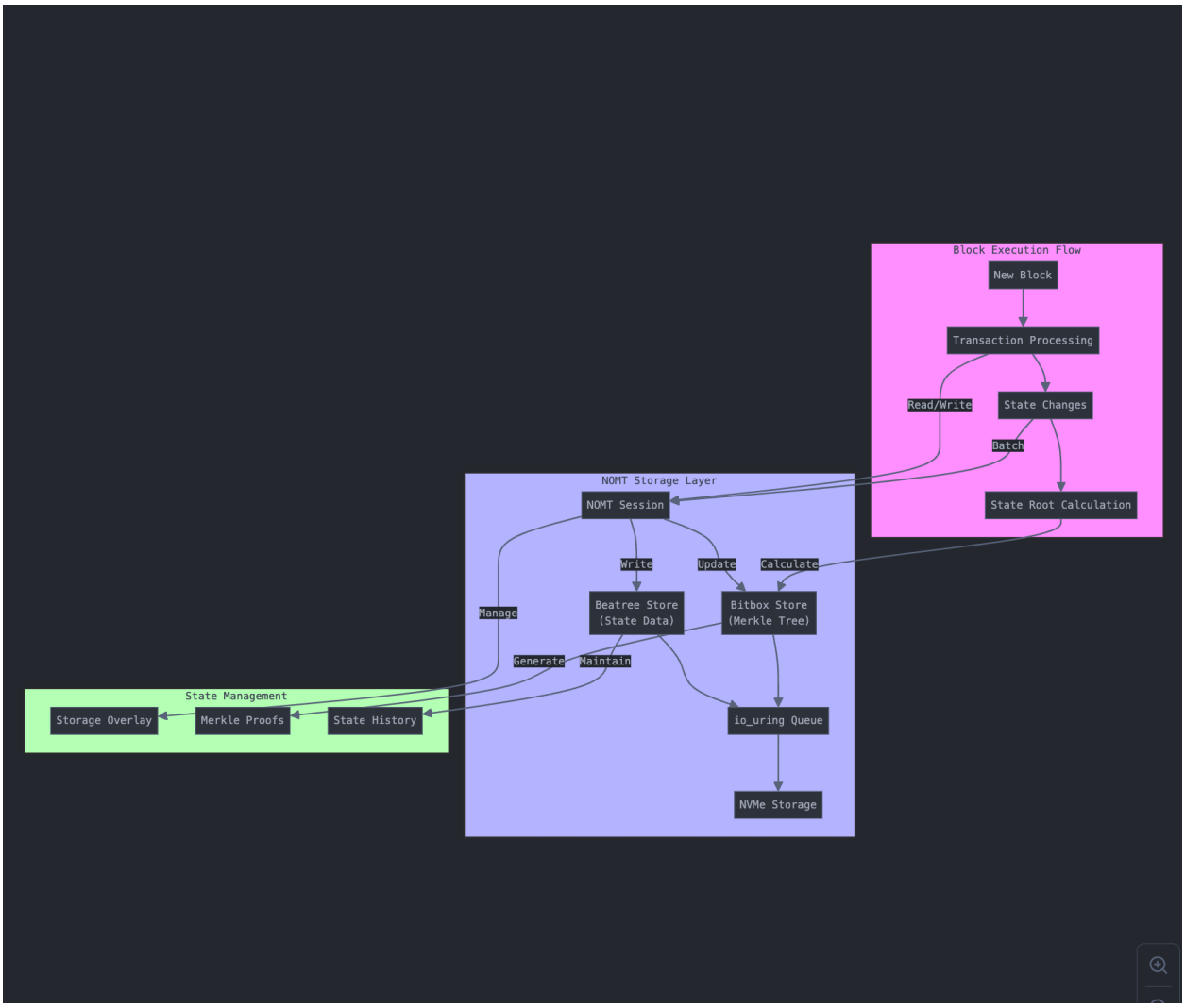
The efficiency gains become particularly apparent during chain synchronization and block import. During these operations, Substrate needs to process multiple blocks in sequence, each requiring state reads, writes, and root hash calculations. NOMT's io_uring-based I/O system allows these operations to be processed asynchronously, significantly reducing the I/O overhead compared to traditional synchronous approaches.
For light client support, NOMT's efficient Merkle proof generation becomes a crucial advantage. Light clients frequently request state proofs, and NOMT's dedicated Merkle tree store can generate these proofs efficiently without needing to reconstruct trie paths from the base key-value store. This separation of concerns allows for optimization of both normal state access and proof generation paths.
State pruning, which is essential for managing blockchain storage growth, benefits from NOMT's direct control over storage layout. The system can more efficiently manage state history and remove outdated states while maintaining access to necessary historical data. This control extends to handling chain reorganizations, where temporary state changes might need to be discarded or reapplied.
The performance impact becomes most noticeable during high-throughput operations. When Substrate needs to process a large number of transactions in a block, or when rapid block production is required, the ability to batch operations

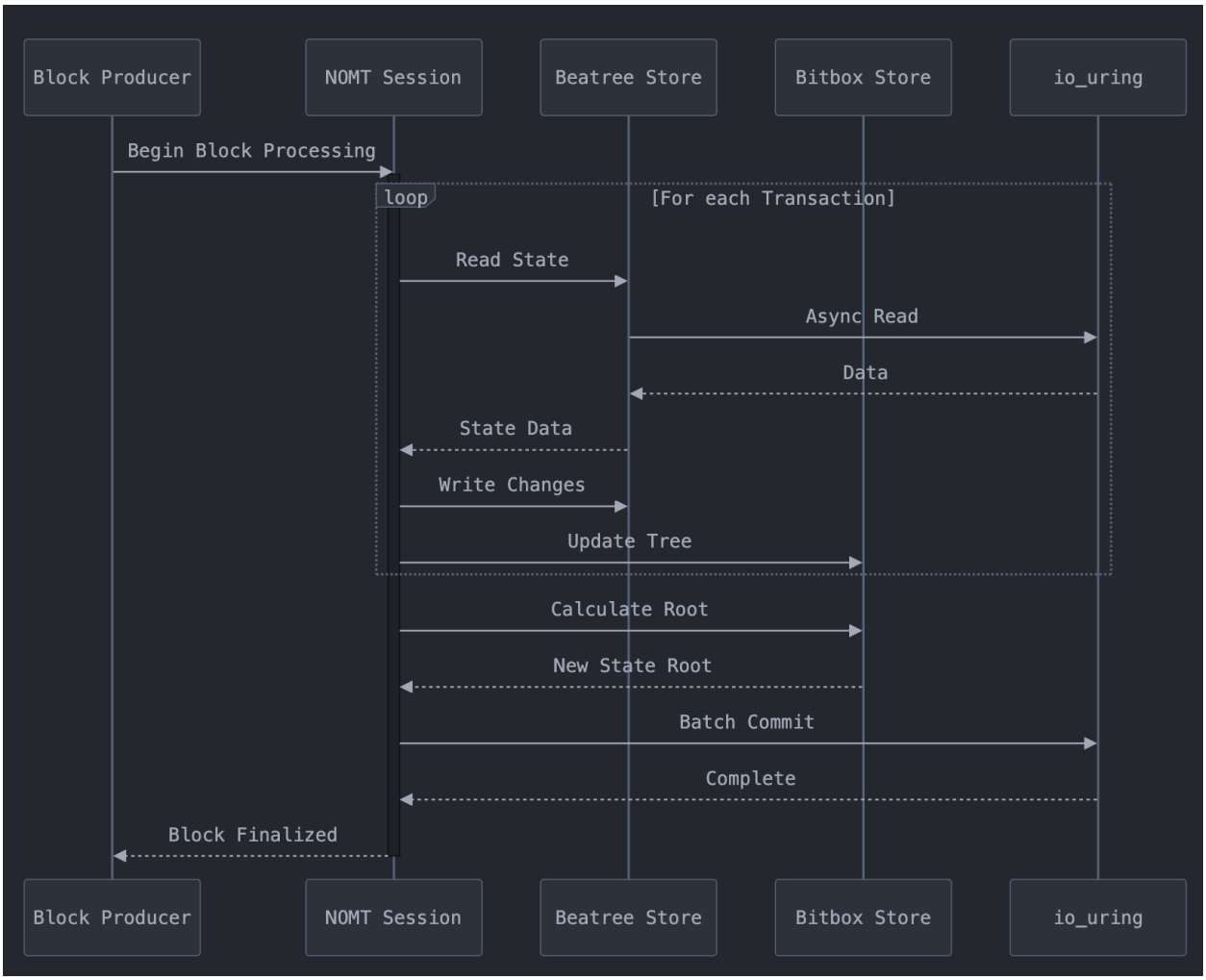
and control I/O patterns directly leads to significant performance improvements. The reduction in system call overhead and better utilization of modern NVMe capabilities means that storage operations no longer become a bottleneck during these high-demand periods.

From a developer's perspective, implementing new runtime modules or modifying existing ones becomes more straightforward with NOMT's clear separation of storage concerns. The storage traits in Substrate can be implemented more efficiently, and the systematic approach to state management reduces the likelihood of introducing performance bottlenecks accidentally.

Error handling and recovery scenarios also benefit from this architecture. When issues occur during block processing, the session-based approach makes it clear what state changes need to be rolled back, and the separate handling of Merkle tree state ensures that the blockchain's cryptographic guarantees are maintained even during error recovery.

Looking at the broader ecosystem, NOMT's architecture provides a foundation for future optimizations. As blockchain state sizes continue to grow and performance requirements increase, having direct control over I/O patterns and state management becomes increasingly important. The ability to adapt to new storage patterns and optimize for specific blockchain workloads makes NOMT a compelling choice for Substrate's evolving needs.


Block Execution Flow
New Block
Transaction Processing
Read/Write
State Changes
Batch
State Root Calculation
NOMT Storage Layer
NOMT Session
Write
Update
Calculate
Beatree Store
(State Data)
Bitbox Store
(Merkle Tree)
Manage
Generate
Maintain
State Management
Storage Overlay
Merkle Proofs
State History
io_uring Queue
NVMe Storage


Block Producer
NOMT Session
Beatree Store
Bitbox Store
io_uring
Begin Block Processing
loop
[For each Transaction]
Read State
Async Read
Data
State Data
Write Changes
Update Tree
Calculate Root
New State Root
Batch Commit
Complete
Block Finalized
Block Producer
NOMT Session
Beatree Store
Bitbox Store
io_uring