# Helios: A Lightweight Path-driven Execution Engine for Hyper-Optimized EVM Clients

**PVLDB Reference Format:**
. Helios: A Lightweight Path-driven Execution Engine for Hyper-Optimized EVM Clients. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

## 1 INTRODUCTION

Ethereum and other EVM-compatible blockchains rely on general-purpose virtual machines to execute smart contracts. As transaction throughput and contract complexity increase, execution becomes a primary scalability bottleneck for both full and archive nodes. While consensus and data availability layers have scaled significantly, the sequential execution of complex smart contracts remains a limiting factor for network performance.

To address this bottleneck, recent research has explored various optimization strategies, ranging from Just-In-Time (JIT) compilation to path-driven speculative execution. Systems such as Forerunner and Seer [**? ?** ] record detailed transaction-level traces or maintain substantial shadow state to guide speculative reuse, while EVMTracer and ParallelEVM [**? ?** ] exploit operation-level parallelism. However, these approaches face an inherent trade-off when applied to modern, highly optimized EVM interpreters such as Revm. We refer to this trade-off as a *performance paradox*: when the baseline EVM is already highly optimized, the overhead of tracing, analysis, and artifact management can outweigh the benefits of optimization. Contract-driven JIT approaches [**?** ], while theoretically powerful, introduce security risks such as JIT bombs and complicate the maintenance of gas-semantic equivalence. Conversely, traditional path-driven approaches incur significant memory I/O and analysis overhead. On fast interpreters where baseline execution completes in microseconds, the cost of loading and processing large trace artifacts often exceeds the execution time itself.

This challenge necessitates a path-driven execution engine that remains safe and gas-precise, yet lightweight enough to deliver net gains on top of modern EVM interpreters. A key observation is that existing systems operate at a suboptimal granularity. Transaction-level caching fails to exploit the repetition of individual contract calls across different transactions, while full-state tracing captures far more information than is necessary for effective optimization.


Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

In this paper, we present Helios, a lightweight, path-driven execution engine designed for high-performance EVM clients. Helios addresses the performance paradox through three core design principles. First, it employs *lightweight asynchronous tracing* that records only stack operations and minimal data dependencies, avoiding the overhead of shadowing full EVM state. This tracing runs off the critical path, so that optimization does not delay live execution. Second, Helios implements *frame-level caching*. We observe that call frames exhibit strong path locality across transactions, so that a relatively small set of cached paths can cover a large fraction of execution. By caching optimized paths at the frame level rather than the transaction level, Helios achieves effective reuse with modest storage overhead. Third, by restricting optimizations to static-cost instructions and delegating all dynamic-cost operations to the underlying EVM, Helios preserves gas semantics by construction, without requiring complex compensation logic.

Helios operates in two distinct modes to support diverse node workloads. In *Replay Mode*, targeted at archive nodes, it leverages pre-computed paths to accelerate historical block processing with no speculation overhead. In *Online Mode*, targeted at full nodes, it employs a simple frequency-based mechanism to speculatively optimize hot paths in real time while falling back to the native interpreter on mispredictions.

We make the following contributions:

- **Problem Analysis and Design Insight.** We analyze why existing JIT- and trace-based EVM optimizers struggle to deliver net gains on top of modern interpreters, and identify frame-level reuse with lightweight tracing as a viable design point. We show that a small fraction of frame-level paths accounts for the majority of execution time on mainnet workloads.
- **System Design.** We design and implement Helios, an execution engine that decouples tracing, optimization, and execution. Our architecture integrates with existing EVM clients, leveraging their native state management while accelerating computational logic through an SSA-based optimizer.
- **Evaluation.** We evaluate Helios on Ethereum mainnet workloads. In Replay Mode, Helios achieves a median speedup of 6.60× over the baseline Revm interpreter. In Online Mode, a frequency-based filter yields net speedups on hot blocks while keeping the storage footprint in the kilobyte range per contract. Our SSA optimizer eliminates a large fraction of stack and arithmetic instructions on hot paths, which contributes to these throughput gains.

The rest of this paper is organized as follows. We provide background on EVM execution and prior optimization approaches in §2, then motivate the design of Helios in §**??**. We present the system design in §**??**, evaluate Helios on microbenchmarks and mainnet

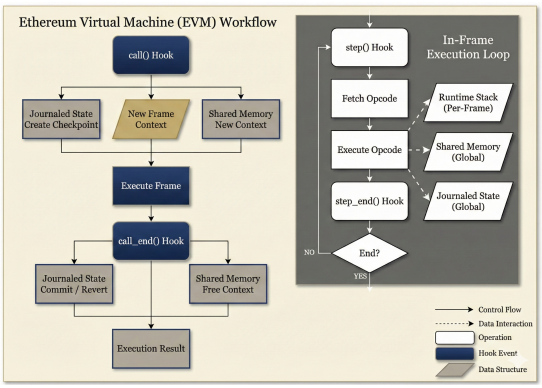


**Figure 1: The Ethereum Virtual Machine (EVM) execution workflow. The process is partitioned into a high-level frame management lifecycle (left) and a low-level, per-opcode execution loop (right). The diagram illustrates the interaction between the runtime stack, volatile memory, and persistent storage, along with the standard hook events utilized by Helios for non-invasive tracing.**

workloads in §??, and discuss implications and future directions in §??.

## 2 BACKGROUND

This section provides the necessary background of the blockchain execution environment. We briefly introduce the core execution model of the Ethereum Virtual Machine (EVM), its resource metering mechanism known as Gas, and the distinct node workloads that motivate the dual-mode design of Helios.

### 2.1 The Ethereum Virtual Machine (EVM) Execution Model

The EVM is a quasi-Turing-complete, stack-based virtual machine that serves as the sandboxed execution environment for smart contracts on the Ethereum blockchain. Its architecture and state model impose specific constraints and present unique opportunities for optimization, which directly inform the design of Helios. The overall execution workflow, depicted in Figure 1, is partitioned into a high-level frame management lifecycle and a low-level, per-opcode execution loop. The following subsections detail the key components of this model.

*2.1.1 Stack-Based Architecture.* The EVM operates on a shared, 1024-element deep runtime stack where each element is a 256-bit word. All computational opcodes retrieve their operands by popping from the top of the stack and push their results back onto it. This model contrasts with traditional register-based architectures that operate on named registers via direct addressing. While conceptually simple, the stack-based design necessitates a high frequency of explicit stack manipulation instructions to manage data flow. These instructions, alongside the overhead of indirect memory access through a stack pointer, represent a primary target for optimization in Helios.

As illustrated in the in-frame execution loop of Figure 1, every operation interacts with this per-frame runtime stack. Stack manipulation instructions, such as DUP1-DUP16 for duplicating items at various depths and SWAP1-SWAP16 for swapping the top element with items at different positions, are prevalent in EVM bytecode as compiler-generated boilerplate for managing operand placement. Unlike computational opcodes, these instructions rearrange existing stack elements without producing new values, making them particularly amenable to elimination.

*2.1.2 State Model: Volatile Memory and Persistent Storage.* The EVM maintains two distinct data storage mechanisms: transient memory and persistent storage, visually represented in Figure 1 as Shared Memory and Journaled State respectively. Memory is a byte-addressable linear array with transaction-scoped lifetime, utilized for ephemeral data such as function call arguments and intermediate computation buffers. Storage, by contrast, implements a persistent key-value mapping from 256-bit keys to 256-bit values, constituting the contract's global state that persists across transaction boundaries. Operations on storage are computationally more expensive than memory operations due to their impact on the global state.

The modification of memory and storage constitutes an observable side effect. This characteristic constrains compiler optimizations, as operations with such side effects cannot be reordered or eliminated without altering program semantics. Consequently, Helios focuses its most aggressive optimizations on the computational, side-effect-free operations that occur between state interactions.

*2.1.3 Call Frames.* EVM execution is organized into a hierarchy of call frames. The lifecycle of a single frame, from its creation to its finalization, is depicted in the workflow on the left side of Figure 1. A new frame, with its own independent memory and runtime stack, is created for each function call, including external calls between different smart contracts. All frames within a single transaction, however, share access to the same persistent storage. A transaction's execution can thus be modeled as an ordered sequence of these call frames.

*2.1.4 Hook Mechanism.* The EVM specification includes a standard hook mechanism to support native tracers and debuggers. This interface allows an external component to subscribe to events that are triggered immediately before and after the execution of each opcode, as well as at the entry and exit points of every call frame, which are explicitly marked as Hook Events in Figure 1. This standard, non-invasive interface is foundational for any external profiling tool, as it enables the passive observation of an execution without modifying the core EVM interpreter.

### 2.2 Gas: The Resource Metering Mechanism

Gas is the fundamental mechanism in the EVM for metering computational resource consumption. It serves both to incentivize validators for the computational work they perform and to protect the network from denial-of-service attacks by ensuring that all executed operations are paid for.

Every transaction submitted to the network must specify a gas limit, representing the maximum amount of gas the originator is willing to consume. Each opcode executed by the EVM deducts a

specific amount of gas from this limit. If the total gas consumed exceeds the limit at any point, the execution is halted, an out-of-gas (OOG) exception is raised, and all state changes made by the transaction are reverted. This mechanism ensures that even programs with infinite loops will eventually terminate, making the otherwise Turing-complete EVM a quasi-Turing-complete machine.

EVM gas costs can be categorized into two types: static and dynamic. Static costs are fixed, compile-time determinable values. The majority of opcodes, such as those for arithmetic or logical operations, have a low, constant gas cost. Dynamic costs are values that depend on the runtime state of the EVM. Examples include the cost of expanding memory, which is a quadratic function of the current memory size, or the cost of an SSTORE operation, which depends on whether a storage slot is being accessed for the first time in the transaction or has been accessed before.

This distinction is central to performance optimization, as it creates the opportunity to aggregate and pre-calculate the cumulative static gas costs of long instruction sequences. Dynamic costs, in contrast, must be calculated at runtime to ensure semantic equivalence.

Certain opcodes, termed gas delimiters, explicitly interact with the gas counter. On healthy execution paths where no out-of-gas exception occurs, these delimiters represent the only points where the gas balance must be observable. GAS reads the current gas balance onto the stack; RETURN, STOP, and REVERT finalize frame execution and require gas verification; CREATE and CREATE2 allocate gas to child contracts. These delimiters partition execution into segments, enabling bulk gas deduction for intervening operations.

## 2.3 Node Types of the Ethereum Network

The dual-mode design of Helios is a direct response to the distinct operational requirements and workload characteristics of the two primary types of nodes in the Ethereum network: Full Nodes and Archive Nodes.

A full node is responsible for participating in the real-time operation of the network. Its primary tasks include receiving new transactions from the peer-to-peer network, executing and validating them, and including them in new blocks. This workload is latency-sensitive, as block production times are fixed. Furthermore, full nodes must process transactions with limited predictability, as they frequently encounter novel smart contracts and previously unseen execution paths. This environment, characterized by low latency requirements and high uncertainty, motivates the need for an execution strategy that can perform on-demand, adaptive optimization as new transactions are observed.

An archive node stores the complete history of the blockchain's state from the genesis block to the present. Its primary workload involves serving historical data queries and, crucially, re-executing large batches of historical blocks, for instance, to sync a new node or for data analysis purposes. This workload is throughput-sensitive, prioritizing the total time to process millions of known, historical transactions over the latency of any single one. The execution is entirely deterministic, as the inputs and outcomes of all historical transactions are already recorded on the blockchain. This high-throughput, deterministic workload motivates the need for a specialized execution strategy that can leverage pre-computed knowledge of historical executions to achieve maximum speed with zero speculation overhead.

## 2.4 Foundational Concepts in Code Optimization

The design of Helios's SSA Optimizer is informed by foundational concepts from the field of compiler theory. A key concept is Static Single Assignment (SSA), an intermediate representation property where each variable is assigned a value exactly once. This property makes data dependencies explicit and simplifies a wide range of optimizations. An execution trace that assigns a unique identifier (e.g., a sequence number) to the result of every operation naturally produces an SSA-like representation. Additionally, the EVM ecosystem presents a clear opportunity for a caching system to implement a code/data separation strategy. Contract standards like ERC-20 result in thousands of deployed instances that share identical logic but differ only in their initial constant values, such as a token's name or total supply. With this background on the EVM's architecture, its resource metering model, and the operational context of blockchain nodes, we now proceed to motivate the design of Helios, addressing the limitations of existing approaches.

## 2.5 Prior EVM Optimization Approaches

Recent systems have explored path-driven optimization strategies to accelerate EVM execution. Forerunner employs constraint-based speculative execution, pre-executing transactions in the transaction pool and generating optimized fast-path programs guarded by control-flow and data-dependency constraints. Seer introduces fine-grained branch prediction with checkpoint-based snapshots to maximize pre-execution result reuse across different execution paths. ParallelEVM achieves operation-level concurrent transaction execution through SSA-based conflict detection and selective re-execution of conflicting operations. EVMTracer provides dynamic analysis tools to construct opcode-level dependency graphs for quantifying parallelization potential and computational redundancy.

These systems employ comprehensive tracing mechanisms that capture complete execution state, including stack operations, memory accesses, and storage dependencies, enabling detailed analysis and optimization of execution paths across EVM-compatible blockchain systems.

# 3 MOTIVATION

Traditional path-driven optimization systems like Forerunner and Seer achieve 5–8× speedups on classical EVM interpreters such as Geth. However, when applied to modern, highly-optimized implementations like Revm, these gains diminish sharply. For identical bytecode, speedups drop from 5.0× on Geth to only 1.5× on Revm. This degradation exposes fundamental limitations in existing optimization strategies when baseline execution is already fast. Through systematic analysis across four critical design dimensions of optimization paradigm, tracing strategy, artifact organization, and correctness guarantees, we derive a set of principled design choices that enable effective optimization on modern EVMs. Each choice addresses specific bottlenecks while establishing constraints

that guide subsequent decisions, ultimately converging on Helios's architecture.

## 3.1 Path-Driven over Contract-Driven Optimization

EVM optimization can follow two competing paradigms. Contract-driven optimization, exemplified by JIT compilation approaches, optimizes entire contracts. Path-driven optimization, employed by systems like Forerunner and Seer, optimizes only executed traces. This foundational choice establishes the scope and safety model for all subsequent design decisions.

Contract-driven approaches compile entire contract bytecode into optimized native code, enabling aggressive global optimizations such as loop hoisting, dead code elimination, and cross-basic-block register allocation. These transformations can yield substantial performance improvements on hot contracts. However, this paradigm faces two critical challenges in production blockchain environments.

First, unrestricted JIT compilation introduces severe security risks through JIT Bomb attacks. Malicious bytecode can exploit compilation complexity by using deeply nested control flow or pathological loop structures to force the JIT compiler into exponential-time compilation or excessive memory consumption. This denial-of-service vector is particularly dangerous in blockchain contexts where attackers can trivially deploy adversarial contracts and force validators to compile them. Existing JIT-based EVM implementations address this threat through contract whitelisting, restricting optimization to pre-approved contracts. This mitigation severely limits applicability, as the majority of contract invocations fall outside whitelists, rendering the optimization ineffective for general workloads.

Second, aggressive global optimizations frequently violate gas-semantic equivalence. Figure 2 illustrates this problem through a representative example where loop hoisting reduces the number of storage reads, thereby changing the gas consumption profile. While this transformation improves performance, it creates economic misalignment between charged fees and actual resource usage. This divergence presents validators with an undesirable choice between overcompensating for work performed or undercompensating relative to network standards, potentially discouraging adoption of the optimization.

Path-driven optimization offers a different trade-off. By optimizing only actually-executed paths rather than entire contracts, this paradigm inherently bounds optimization costs per path, eliminating JIT Bomb vulnerabilities. Every path processed represents real execution that already occurred, ensuring compilation work scales linearly with actual usage rather than potential attack surface. This enables universal deployment without whitelists. Any contract invoking frequently-executed paths automatically benefits from optimization, regardless of bytecode complexity or origin.

However, path-driven systems must still address gas-semantic preservation explicitly. While this paradigm avoids the cross-contract global optimizations that complicate gas preservation in JIT approaches, it does not automatically guarantee gas equivalence. Existing path-driven systems such as Forerunner and Seer do not explicitly discuss this equivalence in their design. Helios achieves precise gas-semantic preservation through specific design choices, particularly the combination of stack-only tracing, frame-level organization, and path-local transformations, as we will demonstrate in §3.4.

**Table 1: Performance degradation of path-driven optimization on modern EVMs. Measurements on Uniswap V2 swap (1-hop) demonstrate that artifact overhead dominates when baseline execution is fast.**

| System | Client | Latency (µs) | Speedup (×) | Artifact (KB) |
|---|---|---|---|---|
| Native | Geth | 398.4 | 1.0× | – |
| Forerunner | Geth | 68.1 | 5.8× | 910 |
| Native | Revm | 62.0 | 1.0× | – |
| Forerunner | Revm | 39.2 | 1.6× | 663 |
| **Helios** | **Revm** | **31.0** | **2.0×** | **70** |

The choice of path-driven optimization establishes our first architectural constraint. We optimize execution traces post-facto rather than contracts a priori. This decision prioritizes safety and universal applicability over maximal theoretical performance, setting the foundation for subsequent design choices.

## 3.2 The Overhead of Existing Path-Driven Approaches

Having established the path-driven paradigm, we now examine why existing implementations struggle on modern EVMs. The core issue is a performance paradox. Detailed tracing, once a minor cost on slow interpreters, becomes the primary bottleneck when baseline execution is already fast.

This paradox manifests most severely in the memory I/O cost of optimization artifacts. Table 1 demonstrates this performance degradation through measurements on a representative Uniswap V2 single-hop swap. Forerunner achieves substantial speedups on Geth but experiences diminished returns on Revm despite generating identical 663 KB trace artifacts. We note that since Forerunner does not provide a native Revm implementation, our measurements employ a functionally equivalent mock implementation that replicates Forerunner's tracing and optimization strategy on the Revm runtime. The reduced baseline execution time on modern EVMs exposes artifact overhead as the dominant bottleneck. When baseline execution completes in tens of microseconds, loading and parsing trace artifacts of this magnitude consumes time comparable to the computational work being optimized.

However, artifact I/O represents only part of the overhead. Full-tracing approaches that maintain custom execution context management—including call frames, memory snapshots, and storage deltas—forgo the host EVM's optimized native implementations. For complex transactions with artifacts exceeding hundreds of kilobytes, I/O latency dominates. For simple transactions such as ERC20 transfers, where artifact sizes remain modest, the computational cost of redundant context management becomes the primary

```
1  PUSH1 0x05
2  JUMPDEST          // loop_start
3  DUP1
4  ISZERO
5  PUSH1 loop_end
6  JUMPI
7    PUSH1 0x01
8    SLOAD
9    POP
10   PUSH1 0x01
11   SWAP1
12   SUB
13 PUSH1 loop_start
14 JUMP
15 JUMPDEST          // loop_end
16 POP
```

**(a) Native execution: 2500 gas**

```
1  PUSH1 0x01
2  SLOAD             // loop hoisting
3  POP
4  PUSH1 0x05
5  JUMPDEST          // loop_start
6  DUP1
7  ISZERO
8  PUSH1 loop_end
9  JUMPI
10   PUSH1 0x01
11   SWAP1
12   SUB
13 PUSH1 loop_start
14 JUMP
15 JUMPDEST          // loop_end
16 POP
```

**(b) After loop hoisting: 2100 gas**

**Figure 2: Loop hoisting optimization violates gas-semantic equivalence. Native execution (a) performs 5 storage reads of slot 0x01 (lines 7–8 in loop body): the first SLOAD incurs a cold read cost of 2100 gas under EIP-2929, while subsequent iterations pay 100 gas each for warm reads, totaling $2100 + 4 \times 100 = 2500$ gas. JIT optimization (b) detects that SLOAD(0x01) is loop-invariant and hoists it outside the loop (lines 2–3), reducing execution to a single storage read consuming only 2100 gas. This 400-gas divergence violates gas-semantic equivalence, creating economic misalignment: charging the original 2500 gas overcompensates miners for actual resource usage, while charging only 2100 gas undercompensates them.**

bottleneck. Modern EVMs provide highly optimized frame-level infrastructure. An effective strategy must leverage these mechanisms rather than duplicating them.

Tracing itself also introduces computational overhead. Existing systems employ different strategies to mitigate this cost. Forerunner and Seer perform tracing in the transaction pool, asynchronously profiling pending transactions before block execution. This removes tracing from the critical path, though artifact I/O costs remain during actual execution. ParallelEVM performs online tracing during parallel execution, where each thread simultaneously executes and traces transactions before resolving conflicts through selective re-execution. This strategy reduces memory I/O by avoiding large pre-generated artifacts and relying instead on runtime dependency tracking.

However, on modern EVMs like Revm, both strategies become prohibitively expensive. We instrumented Revm with a comprehensive tracer recording stack, memory, and storage dependencies during execution. Single-transaction latency increased from 60 µs to over 300 µs, representing a 5× slowdown on the critical path. This overhead stems from per-opcode hook invocations, shadow data structure maintenance, and context switches between the EVM engine and tracing infrastructure. When baseline execution operates at this speed, these mechanisms consume time comparable to the operations being traced.

Furthermore, the purported advantage of full dependency tracking proves illusory in practice. EVMTracer proposed exploiting opcode-level dependencies to parallelize execution within a single transaction. We implemented a dependency-graph-driven parallel executor for Revm that schedules independent opcodes across multiple threads based on recorded dependencies. Contrary to theoretical models, the parallel version consistently underperformed its optimized serial counterpart. The root cause is fundamental. At nanosecond execution granularity, thread synchronization and communication overhead far exceeds the time saved by parallel instruction execution. When individual opcodes complete in 20 nanoseconds but thread coordination requires hundreds of nanoseconds, parallelism merely adds overhead.

**Design Principle 1.** *Minimize analysis overhead through lightweight, asynchronous tracing that leverages host EVM infrastructure.* To achieve meaningful speedups on modern EVMs, an optimization strategy must produce compact artifacts to minimize I/O costs, operate asynchronously off the critical execution path, reuse the host EVM's optimized frame management and state access mechanisms, and capture only information that enables practical, low-overhead optimizations.

This principle establishes the next constraint. Any tracing mechanism must be selective, focusing on the subset of execution state that yields the highest optimization return per byte of trace data. The question then becomes what subset of execution dependencies we should capture.

## 3.3 Frame-Level Reuse over Monolithic Traces

The previous section established the need for lightweight tracing. However, reducing tracing overhead alone is insufficient. We must also maximize the value extracted from each traced execution. This requires addressing two interrelated design decisions regarding the granularity of optimization artifacts and the mechanisms for enabling their reuse across transactions.

Existing path-driven systems adopt transaction-level artifact organization, where each transaction's optimization artifacts are generated and discarded immediately after use. This granularity choice introduces severe limitations for reuse. Consider three transactions where Tx1 calls contracts C1→C2, Tx2 calls C3→C4, and Tx3 calls C1→C3. Under transaction-level organization, Tx3 cannot reuse cached artifacts from Tx1 or Tx2, despite sharing individual contract call frames. The cache key is the entire transaction's execution pattern, the specific sequence C1→C3, rather than individual frame paths. Even though the C1 invocation in Tx3 may be identical to the C1 invocation in Tx1, and the C3 invocation identical to that in Tx2, the system is forced to retrace both frames because

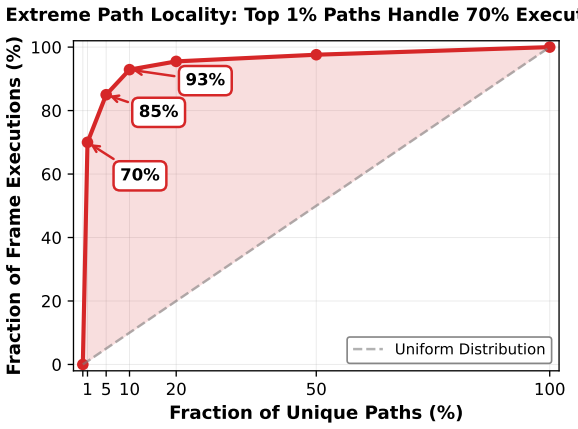


**Figure 3: Path locality in EVM execution exhibits an extreme Pareto distribution. Measured across 5,000 mainnet blocks, the top 1% of unique paths account for 70% of all frame executions, the top 5% handle 85%, and the top 10% handle 93%. The shaded region illustrates the deviation from uniform distribution, demonstrating extreme concentration that enables efficient frame-level caching with minimal storage overhead.**

the transaction-level composition is novel. This use-once-discard model represents significant wasted effort. Even when constituent frames execute along identical paths, the system repeats the full tracing and optimization process.

Frame-level organization resolves this limitation by treating each contract call frame as an independent optimization unit. In the Tx3 example, the individual C1 frame can be recognized and reused from Tx1's cache, and the C3 frame from Tx2's cache. Only genuinely novel frame-level execution patterns require new tracing. This compositional reuse substantially increases effective cache coverage. Rather than requiring exact transaction-level matches, the system benefits from any partial overlap in the frame-level call graph.

To validate this design choice and quantify the economic value of frame-level caching, we conducted large-scale path analysis on Ethereum mainnet blocks 19,600,000 through 19,605,000. For each contract call frame, we compute a path identifier by hashing the sequence of executed opcodes within that frame, then track the frequency distribution of these unique path identifiers across all frames in the analyzed blocks.

The results in Figure 3 reveal a pronounced Pareto distribution with strong path locality. The top 10% of unique paths account for over 90% of all frame executions. Further decomposition shows extreme concentration. The top 1% of paths handle 70% of executions, and the top 5% handle 85%. This distribution remains stable across different block ranges and time periods, indicating that path locality is a fundamental characteristic of EVM execution rather than transient behavior.

This empirical finding confirms the value of persistent, frame-level caching. For high-frequency paths in the top 1%, a single tracing cost can be amortized across hundreds or thousands of subsequent executions. When a path appears 1000 times but requires tracing only once, the effective per-execution overhead becomes negligible, fundamentally changing the cost-benefit ratio compared to transaction-level use-once-discard approaches. The extreme concentration further suggests that even modest cache sizes can achieve high hit rates by retaining only the most frequently executed paths.

Furthermore, frame-level granularity aligns naturally with the EVM's execution model, which inherently manages resources and state at frame boundaries. Each call frame operates within an isolated context with its own memory space, storage view, and gas quota. This alignment allows direct reuse of mature frame management infrastructure from host EVM clients, including optimized memory allocation and storage access patterns. Transaction-level approaches, by contrast, must implement custom frame simulation logic to maintain correctness across artificially flattened execution sequences, adding complexity and potential correctness risks.

The combination of empirical path locality and architectural alignment motivates our second principle.

**Design Principle 2.** *Enable cross-transaction reuse through frame-level, persistent caching.* Optimization artifacts should be organized at the frame granularity and indexed by per-frame path identifiers. This enables compositional reuse across transactions, amortizes optimization costs across high-frequency paths, and leverages host EVM infrastructure for frame management.

This principle establishes two additional constraints. First, artifact organization must respect frame boundaries. Second, caching must support persistent storage with efficient lookup. These constraints, combined with the lightweight tracing requirement from Principle 1, narrow the design space considerably. The remaining question is what specific execution information we should trace and optimize.

## 3.4 Achieving Natural Gas-Semantic Equivalence

The previous sections established the need for lightweight, frame-level, path-driven optimization. However, any optimization strategy must ultimately satisfy a critical correctness requirement specific to blockchain execution. Exact preservation of gas semantics is essential because gas underpins miner incentives in Ethereum's economic model. Transaction senders pay fees proportional to gas consumed. Any optimization altering gas consumption disrupts this economic balance, either overcompensating miners when charging original gas for reduced work or undercompensating them when charging reduced gas. Maintaining gas-semantic equivalence, defined as exact matching of native execution's gas accounting, is thus an economic necessity for practical adoption.

The core challenge stems from the distinction between static-cost and dynamic-cost instructions as defined in §2.2. Static-cost instructions have compile-time-determinable gas costs, while dynamic-cost instructions have runtime-dependent costs that depend on state such as memory expansion or storage access history.

Any optimization that eliminates or reorders dynamic-cost instructions risks gas divergence, as demonstrated in §3.1 with loop hoisting. However, if we restrict optimization to static-cost instructions only, gas preservation becomes straightforward. We precompute the cumulative gas of eliminated operations and deduct it in

bulk, while ensuring all dynamic-cost instructions execute natively with precise accounting.

Remarkably, our previous design decisions, driven by entirely orthogonal concerns, naturally converge to enable this strategy. First, lightweight tracing from Principle 1 led us to focus on operations with low tracing overhead while excluding those requiring substantial state tracking. This scope restriction carries an important secondary consequence. Operations excluded due to tracing cost are precisely those with dynamic gas pricing, such as memory expansion and storage access history, while the traced operations are precisely those with static costs. The excluded instructions also share a common characteristic. They perform side effects on persistent or expandable state, making their costs inherently runtime-dependent. In contrast, the included instructions operate solely on the evaluation stack, a fixed-size structure whose manipulation incurs predictable, compile-time-determinable costs.

Second, frame-level granularity from Principle 2 aligns optimization boundaries with the EVM's native gas accounting boundaries, as each call frame independently tracks gas consumption. Third, the path-driven paradigm from §3.1 constrains optimization to path-local transformations, avoiding cross-basic-block optimizations that would complicate gas accounting across control flow.

The convergence of these three orthogonal choices produces an elegant emergent property. Lightweight tracing naturally selects static-cost operations, frame-level organization aligns with gas boundaries, and path-driven optimization constrains transformations. Together, the optimization scope naturally excludes all dynamic-cost, side-effecting instructions, enabling gas-semantic equivalence without compensatory mechanisms. Each decision, motivated by distinct concerns of overhead reduction, reuse efficiency, and security, contributes guarantees that combine to solve the correctness problem. This is not a coincidence requiring delicate engineering but a robust consequence of constraint composition.

This convergence motivates our third principle.

**Design Principle 3.** *Achieve gas-semantic equivalence as an emergent property of design constraints.* By restricting optimization to static-cost instructions through stack-only tracing, aligning artifact boundaries with gas accounting boundaries through frame-level organization, and constraining transformations through path-driven optimization, gas preservation arises naturally. The system can freely optimize within this scope without trading off performance against correctness.

## 3.5 Synthesis: Deriving the Helios Architecture

The three design principles of lightweight asynchronous tracing, frame-level persistent caching, and emergent gas-semantic equivalence map to Helios's architectural components with varying directness. The first two principles directly drive core components, while the third principle emerges as a natural consequence of their interaction.

Principle 1 manifests in three functional requirements. First, a selective tracer must capture execution dependencies with minimal overhead, generating compact artifacts approximately one order of magnitude smaller than full traces. Second, an asynchronous processing pipeline must decouple tracing from execution, processing artifacts off the critical path through background threads to eliminate online overhead while maintaining low artifact availability latency. Third, the tracing mechanism must leverage the host EVM's optimized native infrastructure for frame management and state access.

Principle 2 necessitates a persistent caching mechanism with per-frame-path indexing. The cache must support both deterministic lookup for known path identifiers and speculative querying for predicted hot paths. Persistent storage enables cross-transaction amortization. High-frequency paths in the top 1% incur tracing costs once but benefit thousands of subsequent executions. Frame-level granularity enables compositional reuse, as demonstrated in §3.3, substantially increasing effective cache coverage compared to transaction-level approaches.

An efficient execution engine is required to consume cached artifacts and accelerate hot path execution. This engine must translate traced execution sequences into an optimized execution model while preserving correctness. Principle 3 constrains the optimization scope to naturally exclude dynamic-cost operations, enabling a straightforward gas accounting strategy. The system precomputes cumulative static gas for optimized operations and deducts it in bulk at frame entry, while delegating all excluded dynamic-cost operations to the native mechanism. This design achieves exact gas equivalence without runtime compensation logic or gas recalculation overhead.

These components support two operational modes addressing distinct node workloads. Replay Mode targets archive nodes performing deterministic historical synchronization. Since all execution paths are known and unchanging, the cache can be pre-populated through one-time tracing of the entire blockchain history. Subsequent replays achieve complete cache coverage, amortizing single tracing costs across thousands of re-executions and enabling substantial speedups with minimal storage overhead, making historical replay economically viable for resource-constrained archive nodes.

Online Mode addresses full nodes processing unpredictable live transactions. Background tracing captures new paths without blocking execution per Principle 1. Dynamic caching maintains coverage of hot paths while evicting cold ones under memory pressure per Principle 2. When a cached path is available, the system executes it optimistically. On cache misses or validation failures, execution falls back to native interpretation, ensuring correctness for never-before-seen patterns.

Critically, both modes share identical core components and artifact formats, differing only in cache population strategy and execution strategy. Pre-fill versus on-demand caching distinguishes Replay Mode from Online Mode, as does deterministic versus speculative execution. This unified design allows a single implementation to span the full node-type spectrum. Quantitative performance evaluation, including detailed speedup measurements, cache hit rates, and storage overhead analysis for both operational modes across diverse workloads, is presented in §**??**.

Building on these three design principles and their instantiation across operational scenarios, the following section presents Helios's detailed architecture.

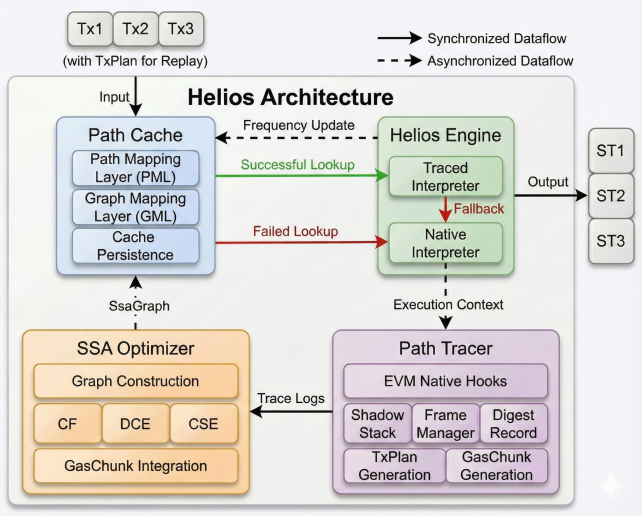


**Figure 4: Overview of the Helios architecture. The system integrates four key components, including Path Cache, Helios Engine, SSA Optimizer, and Path Tracer, to enable path-driven execution. The asynchronous feedback loop (dashed lines) decouples trace generation and optimization from the critical execution path (solid lines), ensuring low-latency transaction processing.**

# 4 THE DESIGN OF HELIOS

## 4.1 Overview

Helios is a path-driven execution engine designed to accelerate EVM transaction processing. Its architecture is built on a continuous feedback loop that integrates four coordinated components: the Path Tracer, the SSA Optimizer, the Path Cache, and the Helios Engine. Notably, path tracing and optimization are performed asynchronously, running in parallel with the primary transaction execution flow. Figure 4 illustrates the system's high-level architecture and data flow. This asynchronous design eliminates optimization latency from the critical path, ensuring that trace generation and graph compilation do not delay transaction processing.

The system's functionality is partitioned across these four distinct components. The Path Tracer captures raw execution traces through lightweight, hook-based instrumentation of the native EVM interpreter, producing a linear execution record known as a PathLog. The SSA Optimizer transforms these PathLog entries into an SSA-like intermediate representation, termed SsaGraph, by applying a pipeline of classic compiler optimizations to eliminate redundant computations. The Path Cache serves as an in-memory memoization layer, indexing and storing optimized SsaGraph structures for rapid, low-latency retrieval. Finally, the Helios Engine orchestrates the execution of each transaction, determining whether to use a cached SsaGraph for accelerated processing or to delegate the task to the native interpreter as a fallback.

## 4.2 End-to-End Transaction Lifecycle

The Helios architecture supports two distinct operational modes—Online and Replay—each defining a different transaction processing lifecycle. The Online mode is designed for real-time processing where execution paths are unknown, while the Replay mode is optimized for high-throughput batch processing of historical transactions with pre-determined paths.

*4.2.1 Online Mode: Executing New Transactions.* Online mode is engineered for the real-time processing demands of full nodes and validators, where execution paths are often unknown. When a function is invoked, the Helios Engine generates an identifier based on the contract and the function being called. It uses this identifier to query the Path Cache for a corresponding SsaGraph. If a matching path is found, the Traced Interpreter begins speculative execution, validated at runtime by control-flow guards. A guard failure or a cache miss triggers an immediate, transaction-level fallback to the Native Interpreter, which executes the transaction to completion to ensure correctness.

This fallback event activates the asynchronous optimization pipeline. The Path Tracer observes the native execution and generates a detailed, linear trace of the operations performed. This trace is then dispatched to the SSA Optimizer, which transforms it into an optimized SsaGraph and its associated constant data. Finally, these new artifacts are stored in the Path Cache, populating it for subsequent executions.

*4.2.2 Replay Mode: Executing Historical Transactions.* Replay mode is tailored for the high-throughput batch processing requirements of archive nodes. In this mode, transaction execution is deterministic. Before processing, the Helios Engine retrieves a pre-computed plan for the transaction, which contains an ordered list of unique identifiers for each execution path. For each frame, the engine uses the corresponding path identifier to perform a direct lookup in the Path Cache.

This lookup is guaranteed to succeed, as the plan only references paths that were successfully traced and cached during a prior execution. The Path Cache returns the precise SsaGraph and its associated constant data. The Traced Interpreter then executes this path directly. In this mode, the speculative nature of the Online mode is entirely eliminated as there are neither cache misses nor control-flow guards needed, and the asynchronous tracing and optimization pipeline is bypassed. This streamlined process enables maximum execution throughput for reprocessing historical blocks.

## 4.3 Key Data Structures

Helios uses a set of data structures to represent, index, and orchestrate the execution of transaction paths. These abstractions manage the system's data flow, enabling both the speculative execution of Online mode and the deterministic processing of Replay mode.

*4.3.1 Path Representation.* Execution paths are captured and optimized through two primary representations:

- **PathLog**: A raw, linear record of an execution path generated by the Path Tracer. It contains a sequence of entries, where each entry details a single operation's opcode and its stack data dependencies, recorded as references to the outputs of prior operations. The PathLog serves as the direct input to the optimization pipeline.
- **SsaGraph**: The optimized, graph-based representation of an execution path. It is a directed acyclic graph where nodes represent operations and edges represent data dependencies. This structure

makes data dependencies explicit, removing the need for a runtime stack machine and serving as the executable format for the Traced Interpreter.

*4.3.2 Path Indexing and Retrieval.* To locate and reuse SsaGraph structures, Helios employs a multi-key indexing scheme:

- **PathDigest**: A 64-bit hash of a path's opcode sequence. It serves as a unique and compact identifier for a specific execution path. Its primary use is for deterministic lookups in the Path Cache.
- **DataKey**: A composite key created by concatenating a contract's code hash with a PathDigest. It is used to index the constant table associated with a specific path in a specific contract instance. This key is necessary to enable a code/data separation strategy. For instance, two different ERC20 token contracts deployed from identical source code will share the same PathDigest for a transfer call. However, their bytecodes may contain different immediate values, such as those encoding the token name or total supply. The DataKey ensures that while they can reuse the same SsaGraph structure, the system retrieves the correct, contract-specific constant table for each.
- **CallSig**: A coarse-grained identifier used for predictive path lookups in Online mode. It is generated by concatenating a contract's code hash with the 4-byte function selector from the transaction's calldata. A single CallSig can map to multiple PathDigests, each representing a distinct control-flow branch that a function invocation may follow, including both successful execution and revert paths.

*4.3.3 Execution Metadata.* Helios captures additional metadata to manage cross-frame execution and optimize performance:

- **Transaction Plan (TxPlan)**: An ordered sequence of PathDigests that records the execution path taken by every function frame within a single transaction. TxPlans are generated during Online mode execution and indexed by block number and transaction index. In Replay mode, they provide the Helios Engine with a deterministic guide for fetching the correct SsaGraph for each frame.
- **GasChunk**: A pre-computed value representing the cumulative static gas cost of a sequence of operations. This sequence is defined as the set of instructions occurring between two specific, pre-defined opcodes that interact with the gas counter. This metadata is attached to the SsaGraph and allows the Traced Interpreter to perform a single, bulk gas deduction instead of per-instruction accounting, reducing overhead while maintaining gas-semantic equivalence.

## 4.4 Component Design and Implementation

This section details the internal design and mechanisms of each of Helios's four primary components. It describes how each component fulfills its role in the end-to-end transaction lifecycle, transforming its inputs into the data structures required by the next stage of the pipeline.

*4.4.1 Path Tracer: Lightweight Execution Path Tracing.* The Path Tracer is the system's instrumentation component, responsible for observing native EVM execution and producing the raw PathLog data structure alongside TxPlan and GasChunk.

**Instrumentation Mechanism.** Path Tracer implements non-invasive instrumentation via the EVM's Hook interface, subscribing to six event types: `step` and `step_end` for opcode-level observation, `call` and `call_end` for external invocations, and `create` and `create_end` for contract deployment. This hook-based design decouples the tracer from the interpreter, enabling passive observation without modifying execution semantics.

**Algorithm 1:** Shadow Stack Tracing

**Input:** $op$: Current EVM opcode
**Input:** $S_{evm}$: EVM value stack (after opcode execution)
**Input:** $S_\ell$: Shadow stack of LSNs tracking value provenance
**Output:** $e$: Trace log entry containing the opcode, current LSN, input dependencies, and output value
**Output:** Updated $S_\ell$

```
// Extract input dependencies from shadow stack
D_in ← []
k ← GetInputCount(op)
for i ← 1 to k do
    ℓ ← S_ℓ.Pop()
    D_in.Append(ℓ)

// Record output value and assign new LSN
if op produces stack output then
    v_out ← S_evm.Top()
    ℓ_curr ← NextLSN()
    S_ℓ.Push(ℓ_curr)
else
    v_out ← ⊥ ;          // No output, e.g., POP, JUMP
    ℓ_curr ← NextLSN()

// Handle stack manipulation instructions
if op ∈ {DUP1, DUP2, ..., DUP16} then
    d ← op − DUP1 + 1
    ℓ_t ← S_ℓ[d] ;                  // Peek without pop
    S_ℓ.Push(ℓ_t) ;                    // Duplicate LSN
if op ∈ {SWAP1, SWAP2, ..., SWAP16} then
    d ← op − SWAP1 + 1
    S_ℓ.Swap(0, d) ;                          // Swap LSNs

// Create log entry
e ← ⟨op, ℓ_curr, D_in, v_out⟩
return e
```

To capture stack data dependencies, the tracer employs a shadow stack. This shadow stack mirrors the EVM's value stack but stores 32-bit Log Sequence Numbers (LSNs) instead of 256-bit values. Each LSN uniquely identifies the operation that produced the corresponding value on the EVM stack.

The tracing process, detailed in Algorithm 1, is driven by the step_end hook, which fires after each opcode execution. For a given opcode, the algorithm first determines the number of required inputs and pops the corresponding LSNs from the shadow stack to form the dependency list. Next, a new LSN is assigned to the current operation. If the operation produces a stack output, this new LSN is pushed onto the shadow stack, tracking the provenance of the new value. Crucially, for stack manipulation opcodes like DUP and SWAP that do not create new values but reorder existing ones, the algorithm mirrors these structural changes on the shadow stack to

maintain a one-to-one correspondence with the EVM's value stack. This process ensures that the generated PathLog entry correctly links each operation to the LSNs of its true data dependencies.

**Metadata Generation.** The Path Tracer is also responsible for generating two key metadata structures: GasChunk and TxPlan. To generate GasChunk metadata, the tracer identifies a set of six gas delimiter opcodes, namely GAS, RETURN, STOP, REVERT, CREATE, and CREATE2. The tracer accumulates the static gas costs of all operations executed between two consecutive delimiters. Upon encountering a delimiter, it finalizes a GasChunk entry containing the accumulated cost. If an execution path terminates without an explicit delimiter, a synthetic STOP is considered to ensure all paths are properly chunked.

The TxPlan for a transaction is constructed using a placeholder mechanism. When a new frame is entered via the call or create hook, a placeholder is appended to the current TxPlan. When the frame execution completes via the call_end or create_end hook, the final, computed PathDigest for that frame replaces the placeholder. This ensures the TxPlan accurately reflects the final sequence of executed paths.

**Path Validation and Filtering.** To ensure that system resources are spent only on reusable paths, the tracer performs a health check. During the call_end hook, it inspects the frame's exit status. Paths that terminate due to VM-level exceptional conditions, specifically out-of-gas errors, are discarded. In contrast, deterministic application-level terminations, including both successful returns and REVERT operations, are retained and cached. This distinction is critical because REVERT paths represent reproducible control-flow branches such as failed require checks or insufficient balance validations, which exhibit high reusability across transactions. VM-level exceptions like OOG, however, exhibit low reusability, as an OOG exception may manifest at any opcode depending on the transaction's gas limit. Recording all possible OOG points would induce exponential path explosion. For a sequence of $n$ opcodes, tracking every potential OOG point generates $O(n)$ distinct failure paths, fragmenting the cache without improving hit rates for deterministic execution. Consequently, only deterministically terminated paths, whether successful or reverted, are formatted into PathLog entries and forwarded to the SSA Optimizer.

*4.4.2 SSA Optimizer: Graph Construction and Optimization.* The SSA Optimizer is a pure-function component that transforms a raw PathLog from the Path Tracer into an optimized SsaGraph and an associated constant table. This process involves four distinct stages.

**Graph Construction.** The optimizer first constructs an initial SsaGraph from the input PathLog. Each entry in the PathLog's sequence is converted into a corresponding node in the graph. Directed edges are then created to represent data dependencies by connecting each node to the predecessor nodes indicated in its $D_{in}$ LSN list. The result is a direct graph-based representation of the linear trace.

**Three-Stage Optimization Pipeline.** The initial graph then undergoes a three-stage optimization pipeline to eliminate computational redundancy: Constant Folding, Dead Code Elimination, and Common Subexpression Elimination. This specific ordering exploits cascading optimization opportunities whereby constant propagation exposes previously unreachable dead code, and the subsequent elimination of dead operations narrows the search space for common subexpression detection.

- **Constant Folding.** The first pass identifies all nodes corresponding to PUSH instructions. Their immediate values are extracted and placed into a constant table. The optimizer then iteratively propagates these constants through subsequent nodes that have no observable side effects, specifically operations that do not modify memory, storage, or system state. Nodes whose computations are fully resolved at compile time are marked as REMOVED, and their outputs are added to the constant table.
- **Dead Code Elimination.** The second pass performs a backward dataflow analysis, starting from nodes that have observable side effects. Any node whose output is not consumed by a non-removed successor is also marked as REMOVED. This process repeats until a fixed point is reached, ensuring that all computationally unnecessary operations are pruned from the graph.
- **Common Subexpression Elimination.** The final pass identifies and merges redundant computations. To do this, it constructs a unique computational fingerprint for each side-effect-free operation. This fingerprint is a deterministic value derived from the operation's opcode and the LSNs of its input nodes. Formally, an operation with opcode $o$ consuming inputs from nodes with LSNs $\ell_i$ and $\ell_j$ yields the fingerprint $\langle o, i, j \rangle$. For example, an `ADD` operation with inputs $\ell_5$ and $\ell_{12}$ produces the fingerprint $\langle \texttt{ADD}, 5, 12 \rangle$. Nodes that share an identical fingerprint are considered redundant and are unified, with all consuming nodes redirected to reference the first canonical occurrence.

  **GasChunk Integration.** Following the optimization pipeline, the optimizer integrates the GasChunk metadata collected by the Path Tracer. It retrieves the list of GasChunks from the PathLog and attaches each pre-computed gas cost to its corresponding delimiter node within the SsaGraph. This annotation embeds the gas accounting information directly into the executable graph structure.

  **Graph Compaction and Output.** In the final stage, the optimizer physically deletes all nodes previously marked as REMOVED to produce a compact graph. It finalizes the constant table, containing all immediate values and folded constants from the optimization phase. The resulting SsaGraph, its constant table, and associated identifiers are then transmitted to the Path Cache for storage.

*4.4.3 Path Cache: Execution Path Memoization.* To efficiently support both the probabilistic lookups of Online mode and the deterministic lookups of Replay mode, the Path Cache employs a two-tier architecture that separates prediction logic from canonical storage.

- **Path Mapping Layer (PML)**: A frequency-based prediction index that supports Online mode's speculative execution. For each CallSig, it maintains a dual-index structure consisting of a frequency map for O(1) frequency queries and a sorted index for O(log k) maximum frequency retrieval, where k denotes the number of distinct frequency values. The PML returns the unique PathDigest with maximum access count for a given CallSig. If multiple paths share the maximum frequency, the lookup returns no prediction, prioritizing prediction accuracy over coverage as low-confidence speculation would incur guard validation overhead.
- **Graph Mapping Layer (GML)**: A deterministic key-value store that serves as the canonical repository for all optimized execution

**Algorithm 2:** Online Mode: Path Lookup

**Notation:** $\mathcal{S}_\sigma$ denotes a PathStore for CallSig $\sigma$, maintaining $M_{freq}$ (PathDigest → frequency map) and $I_{sorted}$ (frequency → PathDigest set, sorted index).

**Input:** $\sigma$: CallSig (code hash ‖ function selector)
**Input:** $h_c$: Contract code hash for DataKey construction
**Input:** $\mathcal{P}$: Path Cache with PML and GML layers
**Output:** $(G, C, \ell)$: Cached graph, constant table, and PathDigest; or $\perp$ if prediction fails

```
// Phase 1: Query Path Mapping Layer for hot path
if σ ∉ P.PML then
    return ⊥ ;          // Cold start: CallSig never observed
S_σ ← P.PML[σ] ;                // Retrieve PathStore
AcquireReadLock(S_σ)
// Get unambiguous maximum frequency path
(f_max, P_max) ← I_sorted.LastEntry()
if |P_max| ≠ 1 then
    ReleaseReadLock(S_σ)
    return ⊥ ;   // Ambiguous: multiple paths share max frequency
ℓ ← P_max.First() ;    // Extract the unique hot path
ReleaseReadLock(S_σ)
// Phase 2: Query Graph Mapping Layer for artifacts
if ℓ ∉ P.GML.graphs then
    return ⊥ ;              // Path not yet optimized
k_data ← h_c‖ℓ ;                  // Construct DataKey
if k_data ∉ P.GML.data then
    return ⊥ ;              // Constant table missing
G ← P.GML.graphs[ℓ]
C ← P.GML.data[k_data]
return (G, C, ℓ) ;             // Successful prediction
```

artifacts. It implements the code/data separation strategy through two distinct mappings. The first maps each PathDigest to its corresponding SsaGraph structure, enabling code reuse across contracts with identical bytecode structure. The second maps each DataKey to a contract-specific constant table, ensuring correct constant retrieval for contracts that share code but differ in embedded immediate values.

**Query Protocols.** The cache's query protocol differs based on the operational mode. In Online mode, a query begins at the PML. The engine provides a CallSig, and the PML retrieves the sorted index for that CallSig under a read lock. It inspects the highest frequency set and returns the associated PathDigest only if it is unique; if multiple paths share the maximum frequency, the query returns no prediction. When a prediction succeeds, the PathDigest is combined with the contract's code hash to form a DataKey, which is then used to query the GML for the SsaGraph and its corresponding constant table. The PathDigest is also returned to the engine for use in the subsequent feedback loop. The complete lookup process is detailed in Algorithm 2.

In contrast, Replay mode bypasses the PML entirely. The engine provides a PathDigest retrieved from a TxPlan and a DataKey. These keys are used to perform a direct, deterministic lookup in the GML to retrieve the SsaGraph and its constant table. For any path referenced in a TxPlan, this lookup is guaranteed to succeed.

**Update and Persistence.** The cache's update protocol distinguishes between two types of events, each serving a different purpose in maintaining the cache's accuracy and coverage. The first type occurs when the SSA Optimizer generates a new SsaGraph, triggering a comprehensive update. The new graph and its constant table are stored in the GML, while the PML initializes or updates the frequency statistics for the corresponding CallSig-PathDigest pair. This mechanism serves as the primary method for populating the cache with new content discovered during native execution.

The second type of update is triggered when the Helios Engine successfully completes a transaction using a cached SsaGraph in Online mode. This event initiates a lightweight feedback update in which the engine signals the PML to increment the access counter for the specific PathDigest that was executed. The PML acquires a write lock on the corresponding frequency statistics and performs an atomic update in O(log k) time: it removes the PathDigest from its old frequency set in the sorted index, increments its frequency in the frequency map, and inserts it into the new frequency set. This feedback loop reinforces the dominance of successful and frequently executed paths, directly improving the accuracy of future predictions. The complete update protocol is formalized in Algorithm **??**.

Because the Path Tracer and SSA Optimizer execute asynchronously relative to the Helios Engine, PML operations are protected by fine-grained read-write locks on a per-CallSig basis to ensure thread safety while maximizing concurrency. Read operations, such as hot path prediction, allow concurrent access, while write operations, such as frequency updates, acquire exclusive locks for bounded time.

**Checkpoint generation and cleanup.** To ensure durability and enable fast node restarts, the system generates checkpoints every N blocks, serializing the cache's contents to disk. The default policy preserves all cached paths, but a pruning mechanism based on CallSig access frequency is provided. The access threshold can be configured to evict CallSigs, along with their associated graphs and data, falling below the specified frequency, trading coverage for storage efficiency.

**Recovery and bootstrapping.** Upon node restart, the Path Cache loads verified high-value paths from checkpoints, immediately achieving high prediction accuracy without a cold-start learning phase. The sorted indices are reconstructed in O(N log k) time during deserialization, where N is the total number of unique paths. The loaded paths represent optimized hotspots from historical execution, validated through frequency and predictability metrics. For paths absent from checkpoints, the system employs lazy regeneration, tracing and optimizing them on-demand during execution to gradually expand path coverage.

*4.4.4 Helios Engine: Dual-Mode Execution Orchestrator.* The Helios Engine is the central orchestrator that coordinates transaction processing. It integrates with the host EVM implementation by replacing its per-frame execution loop while inheriting its mature state and memory management infrastructure. For instance, in its integration with Revm, Helios leverages the host's high-performance memory subsystem, which amortizes allocation overhead by partitioning a pre-allocated byte array across multiple frames via logical checkpointing. Similarly, it inherits Revm's optimized storage access model, which uses in-memory caching and prewarming to avoid redundant disk lookups and a transaction-local journal to buffer state changes for atomic commits. By building upon this robust foundation for state handling, the Helios Engine can focus its design exclusively on optimizing the computational flow within each execution frame. It assumes full control over how opcodes are interpreted, selecting between its two internal interpreters—the Traced Interpreter and the Native Interpreter—based on the operational mode and cache state.

**Transaction-Scoped Execution Model.** A core design principle of the engine is its transaction-scoped execution model. All execution attempts, whether using the Traced Interpreter or the Native Interpreter, operate at the granularity of a full transaction. If any failure occurs during a traced execution—such as a cache miss, a guard violation, or an out-of-gas condition—the engine discards all partial work for that transaction and restarts its execution from the beginning using the Native Interpreter. This strategy eliminates the need for complex, fine-grained state checkpointing or rollback mechanisms, leveraging the EVM's inherent transaction-level atomicity to ensure correctness.

**The Traced Interpreter.** When the Path Cache returns a valid SsaGraph, the engine dispatches execution to the Traced Interpreter. Its execution loop, formally detailed in Algorithm 3, is designed for high-speed graph processing and differs from a standard EVM interpreter in three key aspects.

First, it replaces the EVM's stack machine with a register-based model. As shown in the algorithm, a virtual register file is allocated, and each SsaGraph node's result is stored at an index corresponding to its LSN. An operation's inputs are sourced directly from this array, eliminating all stack manipulation overhead.

Second, in Online mode, it enforces speculative execution guards. The algorithm demonstrates how, before executing a JUMP or JUMPI, the interpreter computes the runtime jump target and validates it against the statically cached target from the SsaGraph. Any mismatch triggers an immediate transaction-level fallback. These guards are disabled in Replay mode.

Third, it implements chunked gas accounting, distinguishing between static and dynamic gas costs. For most instructions, no check is performed. At delimiter nodes, the cumulative static cost from the GasChunk is deducted in a single operation. Any instruction with dynamic gas costs such as memory expansion triggers a separate, per-instruction calculation and deduction. This hybrid approach maintains exact gas-semantic equivalence while minimizing verification overhead.

## 4.5 Security Considerations

Beyond the JIT Bomb resistance established through the path-driven paradigm in §3.1, Helios's design inherently mitigates path explosion attacks. An adversary might attempt to degrade system performance by constructing malicious contracts that generate millions of unique execution paths for a single function signature, potentially flooding the cache and consuming optimization resources.

Helios's architecture naturally defends against this attack vector through three complementary mechanisms. The frequency-based Path Mapping Layer ensures that only paths with demonstrated reusability are predicted and accelerated. Attack-generated paths remain perpetually classified as cold paths due to low execution counts, excluded from the prediction model. The checkpoint pruning mechanism evicts CallSigs with access frequencies below a configurable threshold, preventing malicious cold paths from consuming long-term storage while retaining legitimate hot paths. The transaction-scoped execution model ensures that cache misses simply trigger fallback to the Native Interpreter, maintaining correctness and baseline performance for unpredicted paths.

Consequently, path explosion attacks impose only bounded costs on asynchronous tracing and temporary cache occupancy without degrading performance for legitimate transactions. This robustness emerges naturally from the frequency-based filtering and bounded resource allocation inherent to Helios's design.

# 5 EVALUATION

This section evaluates Helios under both controlled microbenchmarks and real-world mainnet workloads to assess its effectiveness as a path-driven execution engine for EVM optimization. The evaluation aims to answer three core research questions:

**RQ1: Optimization Overhead.** We compare Helios's lightweight path tracing against full contract tracing in terms of time and space cost, assessing whether the overhead remains acceptable for online deployment in production blockchain nodes.

**RQ2: Performance Gains.** We evaluate the speedup achieved by Helios relative to native EVM execution and existing optimization approaches, examining whether the path-driven optimization strategy delivers consistent improvements across diverse smart contract workloads.

**RQ3: System Applicability.** We assess the performance of Replay mode and Online mode in their respective target scenarios—archive node acceleration and full node optimization.

The evaluation proceeds in three stages. §5.1 describes the experimental setup, including hardware configuration, baseline systems, and workload selection. §5.2 presents microbenchmark results for three representative DeFi transactions, isolating the impact of path tracing overhead, execution speedup, and SSA optimization effectiveness. §5.3 analyzes Helios's performance on mainnet blocks, validating path locality assumptions and measuring aggregate throughput improvements.

## 5.1 Experimental Setup

All experiments were conducted on an AWS r7i.2xlarge instance with 8 vCPUs and 64 GB of memory. Helios is implemented in Rust and compiled with release optimizations enabled. We evaluate

against four baseline systems. Geth v1.9.9 serves as the reference Go-based Ethereum client for native EVM execution. Revm v22.0.1 provides a highly optimized Rust-based EVM implementation with substantially faster native performance. Forerunner is tested in two variants: the original Geth-based implementation and a mocked Revm version that replicates its tracing and optimization strategy. Revmc v0.1.0 represents a JIT compilation approach to EVM optimization, evaluated in both native and optimized execution modes. Revmc bundles its own Revm execution environment, which may differ from our standalone Revm v22.0.1 baseline.

## 5.2 Microbenchmark Performance

We evaluate Helios using three representative DeFi transactions of increasing complexity. **ERC20-Transfer** executes 492 opcodes for token transfers, representing the most frequent transaction type on Ethereum. **Uniswap-V2-Swap-1hop** performs single-pool exchanges with 5,667 opcodes, typical of high-liquidity pairs. **Uniswap-V2-Swap-4hop** executes multi-pool routing across 18,063 opcodes, representing complex swap paths for long-tail assets. These benchmarks enable systematic analysis of optimization overhead, execution speedup relative to baselines, and the relationship between opcode reduction and performance gains. Each transaction executes 100 times with warm caches, reporting median values.

*5.2.1 Optimization Overhead.* This subsection addresses **RQ1** by quantifying the optimization overhead in terms of tracing time and artifact storage footprint. Table 2 compares Helios against Forerunner's full-tracing approach on both Geth and Revm. We note that our Forerunner-Revm implementation employs intrusive instrumentation for tracing, while Helios uses a hook-based mechanism that preserves native execution performance.

Helios achieves lower tracing overhead than full-tracing approaches. For ERC20-Transfer, Helios completes path tracing in 23.2 $\mu$s, a 12.5× speedup over Forerunner-Geth. The tracing latency remains comparable to the optimized Forerunner-Revm implementation. As contract complexity increases, the advantage over Forerunner-Geth diminishes to 2.4× for Uniswap-Swap-1hop and 1.4× for Uniswap-Swap-4hop. For complex contracts, the volume of executed instructions dominates tracing cost regardless of strategy.

Storage efficiency gains are more pronounced. Helios achieves 82.7× compression over Forerunner-Geth for ERC20-Transfer, 13.0× for Uniswap-Swap-1hop, and 16.2× for Uniswap-Swap-4hop. Relative to Forerunner-Revm, Helios achieves 9.4× compression for ERC20-Transfer, 9.2× for Uniswap-Swap-1hop, and 16.4× for Uniswap-Swap-4hop. The absolute artifact sizes of 4.8 KB, 70.2 KB, and 122.9 KB enable efficient in-memory caching without memory pressure. For the most complex benchmark, Helios stores the optimized path in 122.9 KB compared to Forerunner-Revm's 2,017.5 KB, a 16.4× reduction that allows hundreds of cached paths to fit within typical L3 cache budgets. This sub-megabyte footprint per contract makes Helios practical for online deployment where storage overhead directly impacts scalability.

With tracing latency in the sub-millisecond range and storage requirements measured in kilobytes, we now evaluate the execution performance gains.

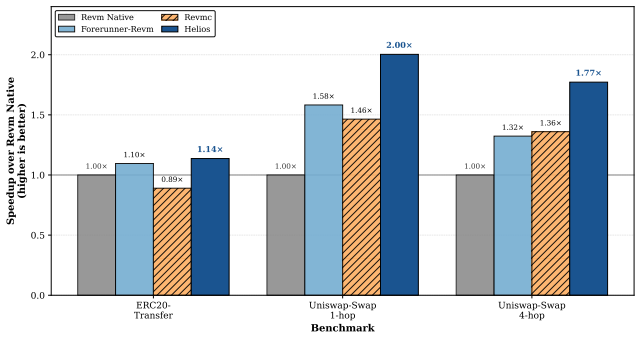


**Figure 5: Execution speedup over Revm 22.0.1 baseline. Higher values indicate greater performance improvement. Revmc results are proportionally scaled from its bundled baseline. Helios achieves the highest speedup across all benchmarks: 1.14× on ERC20-Transfer, 2.00× on Uniswap-Swap-1hop, and 1.77× on Uniswap-Swap-4hop.**

*5.2.2 Execution Performance.* This subsection addresses **RQ2** by evaluating execution speedup on optimized paths. Figure 5 presents acceleration factors for Revm-based systems normalized against Revm Native, which represents state-of-the-art substrate efficiency. Table 3 compares Geth-based and Revm-based implementations.

Helios achieves speedups of 1.14×, 2.00×, and 1.77× for ERC20-Transfer, Uniswap-Swap-1hop, and Uniswap-Swap-4hop. Performance peaks at medium-complexity Uniswap-1hop, where repetitive automated market maker computations create optimization opportunities. Helios reduces execution time from 62.0 $\mu$s to 31.0 $\mu$s by eliminating redundant arithmetic and logic operations through SSA-based path specialization. The modest gain on simple ERC20-Transfer suggests highly optimized baselines leave limited room for interpreter-level optimization.

**Substrate Efficiency Constraint.** Substrate selection fundamentally constrains optimization effectiveness. While Forerunner-Geth reduces Geth Native execution time by 2.5× to 5.8× across benchmarks, optimized Geth execution at 35.75 $\mu$s for ERC20-Transfer remains 7.2× slower than unoptimized Revm at 4.94 $\mu$s. This disparity diminishes on complex workloads, with Forerunner-Geth achieving near-parity on Uniswap-4hop at 167.38 $\mu$s versus 172.49 $\mu$s. Even advanced optimization cannot overcome substrate efficiency gaps on simple workloads. We therefore focus on Revm-based systems to isolate optimization technique effectiveness.

**Lightweight vs. Full-Context Tracing.** Comparing Revm-based systems isolates tracing strategy impact from implementation artifacts. Both Helios and Forerunner-Revm capture optimization opportunities from executed traces at path-level granularity. The key distinction is tracing scope: Helios employs lightweight stack-only tracing, whereas Forerunner performs full-context tracing capturing stack frames, memory snapshots, and contract state. Helios outperforms Forerunner-Revm by 1.04×, 1.27×, and 1.34× across benchmarks. On Uniswap-1hop, Helios achieves 2.00× versus Forerunner-Revm's 1.58×, reducing execution time from 39.20 $\mu$s to 30.96 $\mu$s.

This advantage stems from compact optimization artifacts. Lightweight tracing generates 4.87 KB versus 402.8 KB for full-context tracing. Reduced artifact size enables faster loading and deserialization when replaying optimized paths, while simpler trace structure reduces interpreter overhead by processing fewer metadata fields per instruction. This decreases cache pressure and per-instruction cost. The lightweight approach sacrifices no optimization effectiveness for deterministic execution paths, which constitute the majority of mainnet transactions.

**JIT Compilation Tradeoffs.** Revmc's just-in-time compilation demonstrates mixed effectiveness. For ERC20-Transfer, Revmc underperforms the native baseline by 11% despite using pre-compiled machine code. Several factors likely contribute. Modern interpreters like Revm employ micro-optimizations such as computed gotos, inline caching, and specialized fast paths. These optimizations benefit from whole-program compilation. JIT-generated code, constrained by runtime generation, may fail to achieve comparable efficiency. This limitation becomes particularly significant for short execution sequences where instruction cache effects and setup overhead dominate.

Performance improves on complex workloads to 1.46× and 1.36× for Uniswap-1hop and Uniswap-4hop, suggesting longer execution sequences better amortize fixed overheads. However, Revmc remains inferior to Helios at 2.00× and 1.77× on these benchmarks, with geometric mean speedup of 1.24× versus 1.64×.

These results suggest interpreter-level optimization offers advantages over JIT compilation for EVM workloads. Helios performs SSA transformations on abstract opcode sequences and replays optimized paths through Helios Engine, preserving micro-architectural optimizations in the hand-tuned interpreter while reducing instruction count. JIT compilation replaces the interpreter entirely, requiring runtime code generation to match pre-compiled interpreter efficiency. This appears challenging for microsecond-scale EVM transactions. However, definitive conclusions would require controlled experiments isolating individual optimization components.

To understand the mechanisms underlying these performance gains, we now decompose the impact of individual SSA optimization passes and examine why significant opcode reduction translates to moderate execution speedup.

*5.2.3 Opcode Reduction and Speedup Discrepancy.* This subsection provides deeper insight into **RQ2** by examining the relationship between opcode reduction and execution speedup across the three benchmarks.

**Optimization Effectiveness.** Constant folding dominates opcode reduction, accounting for 97.5%, 97.8%, and 97.9% of eliminated instructions as is shown in Table 4. SSA-based dataflow analysis propagates compile-time constants through execution paths, enabling elimination of computations with known results, unreachable branches, and redundant operations. Common subexpression elimination and dead code elimination contribute 2.1-2.5% combined, indicating limited redundancy after constant propagation. Consistent reduction rates of 76.6-82.3% across varying workload complexities suggest SSA optimization effectiveness is largely invariant to contract scale.

**Understanding the Speedup Discrepancy.** Helios reduces opcode count by 76-82% through SSA-based optimization, yet achieves only 1.14-2.00× speedup rather than the 4.28-5.66× theoretical prediction based on instruction count proportionality. This discrepancy arises from deliberate design constraints that prioritize economic compatibility over maximal performance gains.

Our optimization strategy targets lightweight operations such as stack manipulation, arithmetic, and control flow. These operations execute in nanoseconds individually but account for the majority of eliminated instructions. Constant folding removes these operations systematically, yielding cumulative microsecond-scale savings per transaction that aggregate to millisecond-scale reductions in block processing time across hundreds of transactions per block. Computationally intensive operations such as KECCAK256 and storage accesses remain unoptimized. As established in §**??**, these operations involve dynamic gas metering. We leave these operations unoptimized to avoid gas metering complications.

The SSA graph execution model incurs additional overhead. Each optimized path traversal requires metadata access, register lookups, and dispatch logic. When eliminated opcodes execute in nanoseconds, these graph traversal costs become proportionally significant relative to the savings achieved. The SSA representation trades interpretation overhead for portability and analyzability without requiring JIT compilation infrastructure. The observed 1.14-2.00× speedups demonstrate that substantial opcode reduction yields measurable performance gains within these architectural constraints. §**??** explores further optimization opportunities under the current design framework.

## 5.3 Mainnet Workload Analysis

Having validated Helios's optimization effectiveness on isolated transactions, we now evaluate performance and storage overhead on real-world workloads using 5,000 consecutive mainnet blocks numbered 19,476,587 through 19,481,586. These blocks contain 921,786 total transactions, of which 567,372 are smart contract invocations. We exclude 351,212 pure transfers because they involve no contract execution and fall outside Helios's optimization scope.

We define *execution coverage* as the fraction of contract executions whose PathDigest and DataKey pair matches a cached optimization artifact. Coverage quantifies cache effectiveness by measuring how many executions reuse pre-computed optimizations without re-tracing. This section quantifies storage requirements, validates speedup under production workloads, and analyzes deployment tradeoffs between Replay and Online modes.

*5.3.1 Storage Overhead.* This subsection continues addressing **RQ1** by analyzing storage requirements to assess Helios's practicality for production deployment. We generate optimization artifacts for 5,000 consecutive blocks to ensure cache warmup and path convergence, then evaluate storage-coverage tradeoffs under different caching strategies.

**Baseline Storage Requirements.** Figure 6 shows storage growth for block data and Helios optimization artifacts across 1,000 to 5,000 blocks. Processing 5,000 blocks generates 426 MB of optimization artifacts when caching all unique execution paths without filtering, representing 41.9% overhead relative to raw block data. Storage overhead exhibits sub-linear growth. The first 1,000 blocks incur

**Algorithm 3:** Traced Interpreter Execution

**Input:** $\mathcal{G} = (V, E)$: Optimized SSA graph
**Input:** $C : \mathbb{N} \to \mathbb{U}_{256}$: Constant value table
**Input:** $\Gamma$: Execution context (mutable)
**Input:** $g_{lim}$: Gas limit
**Output:** $\langle \Gamma, g_{used} \rangle$ or $\perp$ (fallback)

```
// Initialize virtual register file and gas counter
```
$\mathcal{R} : \mathbb{N} \to \mathbb{U}_{256} \leftarrow$ new Array$[|V|]$
$g_{rem} \leftarrow g_{lim}$ ; `// Remaining gas initialized to limit`

```
// Execute vertices in topological order
```
**foreach** $v \in V$ *in topological order* **do**
  $\vec{v} \leftarrow \langle\rangle$ ; `// Operand vector`
  **foreach** $\ell \in in(v)$ **do**
    **if** $\ell \in C$ **then**
      $\vec{v} \leftarrow \vec{v} \cdot \langle C[\ell] \rangle$ ; `// Constant operand`
    **else**
      $\vec{v} \leftarrow \vec{v} \cdot \langle \mathcal{R}[\ell] \rangle$ ; `// Register operand`
  `// Control flow guard verification`
  **if** $op(v) \in \{\textsc{Jump}, \textsc{Jumpi}\}$ **then**
    $pc \leftarrow \textsc{ComputeTarget}(op(v), \vec{v}, \Gamma)$
    $\hat{pc} \leftarrow target(v)$ ; `// Cached target from PathLog`
    **if** $pc \neq \hat{pc}$ **then**
      **return** $\perp$ ; `// Guard violation - trigger fallback`
  `// Chunked gas accounting at delimiters`
  **if** $op(v) \in \{\textsc{Gas}, \textsc{Return}, \textsc{Stop}, \textsc{Revert}, \textsc{Create}, \textsc{Create2}\}$ **then**
    $\delta_s \leftarrow chunk(v)$ ; `// Accumulated static gas cost`
    **if** $g_{rem} < \delta_s$ **then**
      **return** $\perp$ ; `// Gas anomaly - verify natively`
    $g_{rem} \leftarrow g_{rem} - \delta_s$
  `// Execute opcode and update execution context`
  $\rho \leftarrow \textsc{Exec}(op(v), \vec{v}, \Gamma)$
  `// Deduct dynamic gas costs`
  **if** $\textsc{IsDynamic}(op(v))$ **then**
    $\delta_d \leftarrow \textsc{DynamicGas}(op(v), \vec{v}, \Gamma)$
    **if** $g_{rem} < \delta_d$ **then**
      **return** $\perp$ ; `// Out of gas`
    $g_{rem} \leftarrow g_{rem} - \delta_d$
  `// Store result in virtual register`
  **if** $\rho \neq \epsilon$ **then**
    $\mathcal{R}[\ell(v)] \leftarrow \rho$ ; `// Map LSN to result value`
  `// Check for execution termination`
  **if** $op(v) \in \{\textsc{Return}, \textsc{Stop}, \textsc{Revert}\}$ **then**
    **return** $\langle \Gamma, g_{lim} - g_{rem} \rangle$
**return** $\langle \Gamma, g_{lim} - g_{rem} \rangle$

**Table 2: Tracing Time and Artifact Storage Overhead**

| Benchmark | Forerunner | | Helios | Reduction | |
|---|---|---|---|---|---|
| | Geth | Revm | | vs Geth | vs Revm |
| *Tracing Time (μs)* | | | | | |
| ERC20-Transfer | 291.6 | 26.3 | **23.2** | 12.5× | 1.1× |
| Uniswap-Swap-1hop | 742.6 | 381.5 | **309.9** | 2.4× | 1.2× |
| Uniswap-Swap-4hop | 1317.7 | 1119.2 | **943.2** | 1.4× | 1.2× |
| *Artifact Size (KB)* | | | | | |
| ERC20-Transfer | 393.4 | 44.7 | **4.8** | 82.7× | 9.4× |
| Uniswap-Swap-1hop | 910.4 | 647.5 | **70.2** | 13.0× | 9.2× |
| Uniswap-Swap-4hop | 1994.3 | 2017.5 | **122.9** | 16.2× | 16.4× |

**Table 3: Execution Time: Geth-based vs. Revm-based Systems**

| System | ERC20 (μs) | Uniswap-1hop (μs) | Uniswap-4hop (μs) |
|---|---|---|---|
| Geth Native | 89.11 | 398.40 | 966.20 |
| Forerunner-Geth | 35.75 | 68.12 | 167.38 |
| Revm Native | 4.94 | 62.02 | 172.49 |

Despite achieving 2.5-5.8× speedup over Geth Native, Forerunner-Geth (35.75 $\mu s$) remains 7.2× slower than unoptimized Revm Native (4.94 $\mu s$) on ERC20-Transfer, demonstrating that substrate efficiency dominates optimization gains in determining absolute performance.

**Table 4: Opcode Reduction and Execution Speedup Across Benchmarks**

| Metric | ERC20 | 1hop | 4hop |
|---|---|---|---|
| Native opcode count | 492 | 5,667 | 18,063 |
| *Opcodes Eliminated* | | | |
| CF | 395 | 4,249 | 13,604 |
| CSE | 10 | 85 | 257 |
| DCE | 0 | 9 | 33 |
| Total eliminated | 405 | 4,343 | 13,894 |
| Reduction rate | 82.3% | 76.6% | 76.9% |
| **Execution speedup** | **1.14×** | **2.00×** | **1.77×** |
| *Naive prediction* | *5.66×* | *4.28×* | *4.33×* |

CF: Constant Folding; CSE: Common Subexpression Elimination; DCE: Dead Code Elimination.

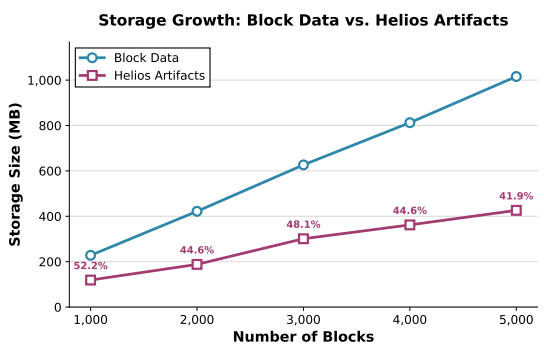


**Figure 6: Storage growth for block data versus Helios op-**

**Table 5: Storage-Coverage Tradeoff for Mainnet Blocks**

| Freq Threshold | Storage | Overhead | Exec Coverage | Top-1 Coverage |
|---|---|---|---|---|
| ≥1 (all paths) | 426 MB | 42.0% | 100.0% | 62.2% |
| ≥10 | 50 MB | 4.9% | 96.5% | 58.0% |
| ≥50 | 38 MB | 3.7% | 90.0% | 52.2% |
| ≥100 | 19 MB | 1.9% | 85.6% | 48.6% |
| ≥500 | 4 MB | 0.3% | 69.9% | 38.4% |

52.2% overhead, decreasing to 41.9% at 5,000 blocks due to path convergence.