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

Sipeng Xie[1], Ruiqi Zhang[1], Minghang Li[1], Qianhong Wu[1], and Bo Qin[2]
[1]Beihang University, Beijing, China [2]Renmin University of China, Beijing, China
{sipengxie, ruiqizhang, minghangli, qianhong.wu}@buaa.edu.cn
bo.qin@ruc.edu.cn

ABSTRACT

Smart contract execution remains a primary scalability bottleneck for Ethereum Virtual Machine (EVM) blockchains. Existing optimization strategies, such as Just-In-Time (JIT) compilation and path-driven speculative execution, face a "performance paradox" where the overhead of detailed tracing and artifact management often negates performance gains on modern, highly optimized clients like Revm. We present **Helios**, a path-driven execution engine designed to resolve this trade-off through lightweight asynchronous tracing and persistent frame-level caching. Unlike transaction-level approaches, Helios exploits the strong path locality of individual contract calls to maximize reuse across different transactions, transforming raw traces into Static Single Assignment (SSA) graphs off the critical path. By restricting optimization to static-cost instructions, the system guarantees gas-semantic equivalence by construction, eliminating the economic risks associated with aggressive compilation. Evaluation on Ethereum mainnet workloads demonstrates that Helios achieves a median speedup of 6.60× over the state-of-the-art client Revm in Replay Mode, making it ideal for archive node acceleration. In Online Mode, frequency-based filtering enables effective acceleration of hot paths with negligible storage overhead. These findings establish Helios as a scalable, safe foundation for next-generation EVM infrastructure.

PVLDB Reference Format:
Sipeng Xie[1], Ruiqi Zhang[1], Minghang Li[1], Qianhong Wu[1], and Bo Qin[2]. 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

Public blockchains supporting smart contracts, such as Ethereum and its Layer-2 rollups [6, 7, 15, 27, 34], fundamentally operate as replicated state machines. From a data management perspective, each node functions as a deterministic transaction processor that must handle two distinct workloads, namely the real-time processing of incoming transactions and the re-execution of historical transactions to verify global state transitions. As network throughput and contract complexity continue to increase, execution efficiency emerges as a primary scalability bottleneck for both scenarios.

Achieving meaningful acceleration on modern, hyper-optimized execution engines [1, 3] requires overcoming three fundamental tensions. **First**, optimization conflicts with semantic preservation. Standard compiler optimizations such as instruction reordering inherently alter the observable execution trace, violating the strict gas accounting rules that underpin economic consensus [2, 5, 9, 11]. A single gas discrepancy can cause nodes to diverge on state validity. Moreover, Just-In-Time (JIT) compilation introduces security vulnerabilities where maliciously crafted contracts exploit compilation overhead to mount denial-of-service attacks [32]. These constraints demand that any viable acceleration strategy preserve gas semantics by construction. **Second**, instrumentation overhead conflicts with execution latency. On sub-microsecond engines such as Revm [1], the cost of synchronous tracing frequently exceeds execution latency itself, creating what we term the *Performance Paradox* [14, 35]. Similarly, operation-level concurrent execution incurs synchronization costs that outweigh parallel gains [26, 28]. Any optimization blocking the critical path risks performance regression. **Third**, artifact granularity conflicts with reuse capability. Transaction-level caching suffers from combinatorial path explosion, as the space of possible paths grows exponentially with contract complexity. This results in ephemeral artifacts with negligible reuse across diverse workloads [14, 35].

To assess whether these barriers can be surmounted, we analyzed Ethereum mainnet workloads and derived three critical insights. **First**, dynamic state-access operations are relatively sparse along hot paths, while fixed-cost computational instructions dominate. Since optimizing dynamic operations risks altering the gas schedule, restricting acceleration to static-cost instructions enables correctness by construction. **Second**, dependencies relevant for optimization are confined to stack operations. Capturing full memory and storage state incurs prohibitive overhead while providing negligible utility, and thus lightweight stack-only tracing suffices for effective optimization. **Third**, the top 1% of unique frame-level paths accounts for over 70% of total execution time, indicating that shifting granularity from transactions to call frames unlocks high cache hit rates across diverse workloads.

Guided by these insights, we present Helios, a path-driven execution accelerator built on three architectural principles. To guarantee safety, Helios employs a hybrid execution model that restricts optimization to static-cost instructions while delegating dynamic-cost operations to the native engine, preserving exact gas semantics by

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

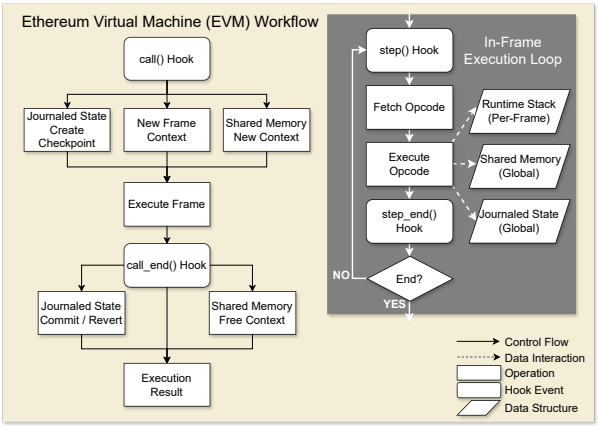

Figure 1: The EVM execution workflow. The process is partitioned into a high-level frame management lifecycle (left) and a low-level, per-opcode execution loop (right).

construction. To break the performance paradox, Helios decouples optimization from the critical path via asynchronous lightweight tracing that captures only stack-level dependencies. To maximize reuse, Helios organizes artifacts at frame-level granularity, enabling diverse transactions to composably reuse cached paths and converting ephemeral transaction-level caching into durable components.

In summary, this paper makes the following contributions:

- We systematically analyze existing acceleration strategies on modern execution engines and formulate the *Performance Paradox*. We identify the separation of static and dynamic costs, stack-confined dependencies, and frame-level locality as key levers to resolve this paradox.
- We propose Helios, a novel architecture achieving correctness by construction through hybrid execution, breaking the performance paradox via asynchronous tracing, and maximizing reuse through frame-level caching.
- We develop a high-performance engine featuring a register-based interpreter and bulk gas deduction, serving as the unified runtime for both deterministic replay and speculative execution.
- We implement Helios on Revm and evaluate it on Ethereum mainnet workloads, demonstrating a median speedup of 6.60× over the native baseline while maintaining exact gas consistency.

2 BACKGROUND & MOTIVATION

2.1 The EVM Execution Model

The Ethereum Virtual Machine operates as a quasi-Turing-complete stack machine [34]. Unlike register-based architectures, the EVM performs all computations on a transient runtime stack using manipulation instructions such as DUP and SWAP to manage operand placement. This design necessitates frequent stack operations that incur significant execution overhead.

Execution is compartmentalized into a hierarchy of call frames. Each frame maintains an isolated memory context and stack while sharing persistent storage access with other frames in the transaction. Resource consumption is metered via gas, which functions as both a validator incentive and a security mechanism against denial-of-service attacks. Gas costs fall into two distinct categories. Static costs are fixed at compile time for computational operations such as arithmetic and stack manipulation. Dynamic costs depend on runtime state, including memory expansion extent and storage access frequency. This distinction is fundamental to our design, as it enables optimization strategies that aggregate static costs while delegating dynamic accounting to native handling logic.

The EVM specification includes a standard hook mechanism to facilitate debugging and tracing. As illustrated in Figure 1, this interface triggers events at key lifecycle points such as opcode execution and frame transitions. This design enables external components to passively observe execution states and capture data dependencies without requiring invasive modifications to the core interpreter logic.

2.2 Ethereum Node Workloads

The Ethereum network depends on node archetypes that exhibit divergent operational characteristics [18]. Full nodes operate within fixed block intervals and prioritize the low-latency processing of unpredictable transactions to maintain consensus stability. Archive nodes maintain comprehensive ledger history and emphasize execution throughput to facilitate bulk re-execution of historical states [17, 21, 25].

This operational dichotomy has direct implications for execution acceleration. Full nodes operate in an online mode where execution paths are unknown prior to invocation, precluding ahead-of-time compilation. Acceleration in this context demands minimal response latency while simultaneously generating optimization artifacts for future reuse. Archive nodes operate in a replay mode where execution paths are deterministic and known in advance, enabling aggressive precomputation. Acceleration in this context demands maximum throughput by leveraging cached artifacts across millions of historical transactions. Any unified acceleration framework must therefore accommodate both modes, balancing low-latency responsiveness with high-throughput batch processing.

2.3 Limitations of Contract-Level Optimization

Contract-level optimization, exemplified by JIT compilation, faces two critical limitations in permissionless blockchain environments. **First**, JIT compilation introduces security vulnerabilities. Attackers can construct pathological code patterns, such as deeply nested conditional branches, to trigger exponential compilation complexity. This computational asymmetry allows malicious actors to exhaust validator resources via low-gas transactions, constituting a denial-of-service vector known as JIT bombs [2, 5, 9, 11, 32]. **Second**, JIT compilation risks gas-semantic discrepancies. Aggressive compiler optimizations such as instruction reordering and dead code elimination alter the observable execution trace, violating the strict gas accounting rules that underpin economic consensus. Any systematic deviation from the canonical gas schedule results in incorrect validator compensation and impedes the deployment of production clients [2, 11].

These constraints establish that contract-level compilation is fundamentally unsuitable for permissionless execution environments. Path-driven optimization emerges as a safer paradigm by

Table 1: Performance degradation of path-driven optimization on modern EVMs.

System	Client	Latency	Tracing	Speedup	Artifact
Native	Geth	398.4μs	–	1.0×	–
Forerunner	Geth	68.1μs	742.6μs	5.8×	910KB
Native	Revm	62.0μs	–	1.0×	–
Forerunner	Revm	39.2μs	381.5μs	1.6×	663KB
Parallel	Revm	417.7μs	–	0.2×	–

restricting acceleration to actually executed paths, thereby bounding optimization costs to the runtime trace and inherently avoiding dead code and unreachable branches. However, existing path-driven approaches encounter their own scalability barriers on modern execution engines.

2.4 The Performance Paradox

Existing path-driven schemes face a phenomenon we term the *Performance Paradox*. On slower engines such as Geth [10], transaction execution dominates end-to-end latency, rendering additional instrumentation costs negligible. A prior path-driven system [14] achieves a 5.8× speedup on Geth. However, on highly optimized engines such as Revm [1], execution becomes inexpensive and auxiliary overhead emerges as the primary bottleneck. The same system achieves only 1.6× on Revm. Table 1 quantifies this shift using a representative Uniswap V2 swap transaction [4].

Three categories of auxiliary overhead contribute to this paradox. **First**, synchronous tracing dominates the critical path. On Revm, capturing full execution state is approximately six times slower than native execution and nearly ten times slower than optimized execution. Any synchronous instrumentation on the critical path effectively negates the speedup from optimization. **Second**, artifact management incurs substantial fixed costs. Loading and parsing a 663 KB artifact to accelerate a task completing in tens of microseconds introduces I/O latency that does not scale with execution time. **Third**, fine-grained concurrency proves counterproductive. We implemented a dependency-graph-driven parallel executor on Revm to evaluate instruction-level parallelism. As shown in Table 1, this approach achieves only 0.2× the throughput of sequential execution. The root cause is granularity mismatch. Individual opcode execution requires approximately 20 nanoseconds, while thread synchronization and context switching require hundreds of nanoseconds [26, 28]. Coordination overhead consistently exceeds parallel gains at this granularity.

These observations collectively establish that synchronous tracing, heavyweight artifacts, and fine-grained parallelism are incompatible with high-performance execution engines. Effective acceleration requires asynchronous optimization decoupled from the critical path, lightweight trace representations, and sequential execution of optimized code.

2.5 The Granularity Mismatch

Beyond auxiliary overhead, existing path-driven strategies predominantly employ transaction-level caching [14, 28, 35]. This approach

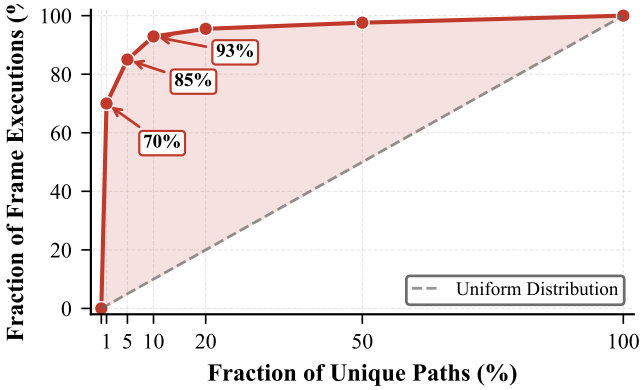

Figure 2: Path locality in EVM execution follows an extreme Pareto distribution: the top 1% of unique paths account for 70% of frame executions (5,000 mainnet blocks). The shaded region shows deviation from uniform distribution.

treats the entire transaction execution trace as the atomic optimization unit. It faces a combinatorial challenge because slight variations in internal call sequences render the entire cached artifact invalid. This dependency enforces a use-once-discard lifecycle that severely limits reuse potential.

Our analysis of Ethereum mainnet workloads reveals significant redundancy at finer granularity. As shown in Figure 2, the distribution of unique frame-level paths follows a Pareto pattern. The top 1% of paths accounts for over 70% of total execution time. This concentration indicates that while unique transaction combinations are vast, individual contract calls function as repetitive building blocks. Shifting the optimization unit from transactions to frames aligns the caching strategy with this inherent locality and enables amortization of compilation costs across thousands of invocations.

2.6 Design Implications

The preceding analysis establishes three architectural principles for high-performance EVM acceleration. **First**, the distinction between static and dynamic gas costs motivates a hybrid execution model. Restricting optimization to static-cost instructions while delegating dynamic operations to native handling ensures gas-semantic equivalence by construction. **Second**, the Performance Paradox necessitates asynchronous lightweight tracing. Decoupling trace generation from the critical path and minimizing trace volume preserves baseline execution speed while enabling background optimization. **Third**, pronounced frame-level locality motivates persistent caching at this granularity. Frame-level artifacts enable compositional reuse across diverse transactions, overcoming the ephemeral nature of transaction-level approaches. These principles collectively inform the design of Helios.

3 THE DESIGN OF HELIOS

3.1 Overview

Helios is a path-driven execution engine designed to accelerate EVM transaction processing. Its architecture comprises four coordinated components. The Helios Engine orchestrates execution. The

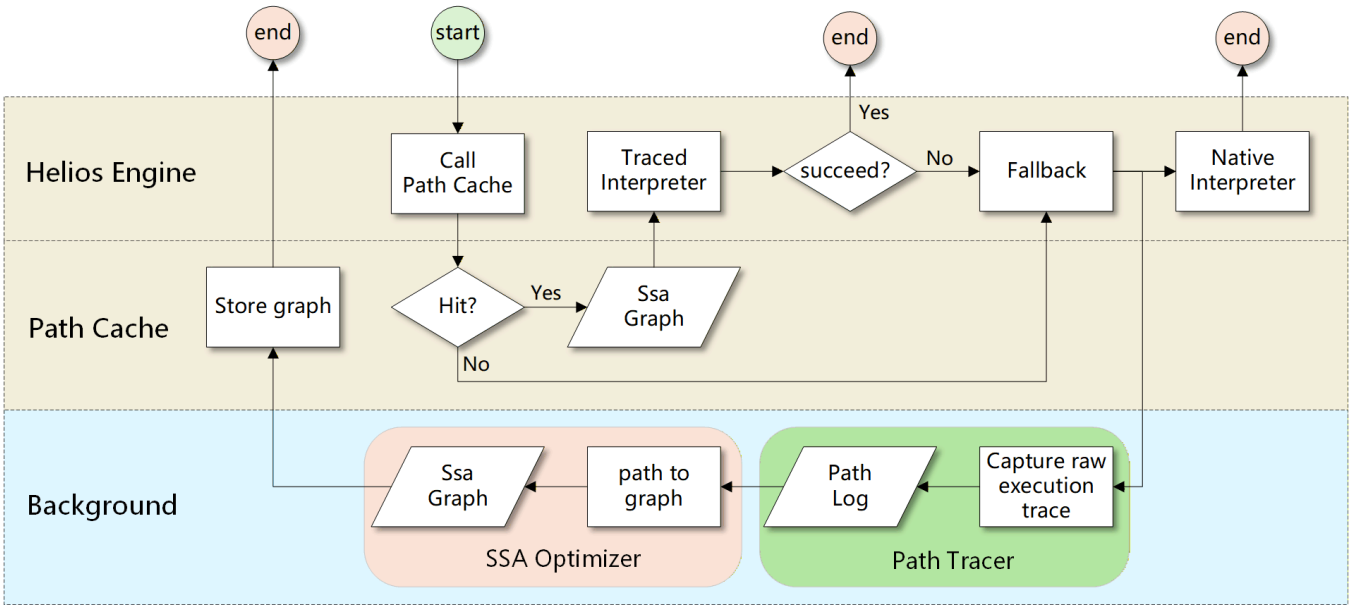

Figure 3: Overview of the Helios architecture.

Path Cache indexes optimized graphs for reuse. The Path Tracer captures execution traces. The SSA Optimizer compiles them into an optimized intermediate representation. Figure 3 illustrates the system architecture and data flow.

Upon contract invocation, the Helios Engine queries the Path Cache using a path identifier derived from the contract identity and call signature. On a cache miss, the engine delegates execution to the Native Interpreter. This delegation simultaneously initiates an asynchronous optimization pipeline. The Path Tracer captures the raw execution trace as a PathLog, a linear sequence of executed opcodes with their data dependencies. The SSA Optimizer transforms this PathLog into an optimized SsaGraph, a directed acyclic graph that makes data flow explicit. The Path Cache then stores the SsaGraph for future reuse. Running in parallel with the Native Interpreter, this pipeline ensures that optimization never blocks the critical execution path.

On a cache hit, the engine retrieves the corresponding SsaGraph and dispatches it to the Traced Interpreter, which performs speculative execution validated by runtime control-flow guards. An execution failure triggers a fallback to the Native Interpreter and re-initiates the optimization pipeline. Successful execution concludes the transaction with reduced overhead.

Helios supports two operational modes. Online mode targets full nodes and validators handling transactions with unknown paths, where cache misses and guard failures may occur. Replay mode targets archive nodes reprocessing historical blocks. A pre-computed TxPlan guarantees cache hits for every call frame, allowing the engine to bypass control-flow guards and the optimization pipeline for maximum throughput.

3.2 Key Data Structures

Helios represents, indexes, and orchestrates transaction paths through a small set of data abstractions that govern data flow in both Online and Replay modes.

Path Representation. Execution paths are first captured as a *PathLog*, a raw linear trace produced by the Path Tracer. A PathLog is a sequence of entries, each recording an opcode together with its stack data dependencies, expressed as references to the outputs of preceding operations. The optimization pipeline consumes this representation and transforms it into an *SsaGraph*, a directed acyclic graph whose nodes denote operations and whose edges denote data dependencies. By making data flow explicit and eliminating the implicit EVM stack, the SsaGraph serves as the executable format for the Traced Interpreter.

Path Indexing and Retrieval. Helios employs a multi-key scheme to locate and reuse SsaGraphs. A *PathDigest* is a 64-bit hash of a path's opcode sequence acting as a deterministic identifier for the execution logic. A *DataKey* concatenates a contract's code hash with the PathDigest to uniquely identify the constant table for a specific contract instance. This separation allows multiple contracts with identical bytecode, such as distinct ERC20 [33] tokens, to share a single SsaGraph while maintaining separate constant tables. Finally, a *CallSig* is a coarse-grained identifier used for predictive lookup in Online mode, defined as the concatenation of a contract's code hash and the 4-byte function selector from calldata. A single CallSig may map to multiple PathDigests, corresponding to distinct control-flow branches of the same function, including both successful and revert paths.

Execution Metadata. Helios maintains additional metadata to coordinate cross-frame execution and reduce runtime overhead. A *Transaction Plan (TxPlan)* is an ordered sequence of PathDigests

Algorithm 1: Shadow Stack Tracing

Input: op: Current EVM opcode
Input: S_{evm}: EVM value stack (after opcode execution)
Input: S_ℓ: 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_ℓ

```
// 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
```

that records the path taken by each call frame within a transaction. TxPlans are produced during Online execution and indexed by block number and transaction index; in Replay mode, they provide a deterministic guide for fetching the correct SsaGraph for every frame. A *GasChunk* is a precomputed scalar capturing the cumulative static gas cost of the instructions lying between two consecutive gas-accounting opcodes. Attached to the corresponding SsaGraph, GasChunks allow the Traced Interpreter to replace per-instruction gas accounting with a single bulk deduction, thereby reducing overhead while preserving gas-semantic equivalence.

3.3 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.

3.3.1 Path Tracer. A lightweight instrumentation component observes native EVM execution to produce the raw PathLog data structure, TxPlan, and GasChunk.

Instrumentation mechanism. The tracer attaches to the EVM hook interface and subscribes to six events, including `step` and `step_end` for opcode execution, `call` and `call_end` for external calls, and `create` and `create_end` for contract creation. This hook-based design decouples the tracer from the interpreter and enables passive observation without changing execution semantics.

To capture stack data dependencies, the tracer maintains a shadow stack mirroring the EVM operand stack but storing 32-bit Log Sequence Numbers or LSNs instead of 256-bit values. Each LSN identifies the operation producing the corresponding value. As detailed in Algorithm 1, the tracer pops the required input LSNs to form the dependency list D_{in} on each `step_end`, allocates a new LSN for the current opcode, and pushes it if the opcode produces a stack result. For stack-manipulation opcodes such as `DUP` and `SWAP` that only reorder values, the tracer applies the same permutation to the shadow stack without creating a new LSN. As a result, only opcodes that actually produce new values become nodes in the PathLog, effectively collapsing substantial stack traffic and yielding PathLog entries that record each operation alongside the LSNs of its true data dependencies.

PathDigest Calculation. PathDigest is a rolling hash updated with each executed opcode using the lightweight FNV-1A algorithm [22, 23]. FNV-1A was chosen for its efficiency, involving simple multiplication and XOR operations. This ensures a unique identifier for each execution path and allows fast incremental updates, enabling efficient path comparison and lookup in the PathCache.

Metadata generation. The tracer also constructs GasChunk and TxPlan metadata. For GasChunks, it treats `GAS` and terminating opcodes such as `RETURN`, `STOP`, `REVERT`, `CREATE`, and `CREATE2` as gas delimiters. It accumulates the static gas cost of instructions between two delimiters and emits a GasChunk with the aggregate cost upon reaching a delimiter. If a path ends without an explicit delimiter, the system inserts a synthetic `STOP` to close the final chunk.

The TxPlan is built using placeholders. When a new frame is entered via the `call` or `create` hook, the tracer appends a placeholder entry. When the frame completes at `call_end` or `create_end`, it computes the frame's PathDigest and replaces the placeholder. The resulting TxPlan records the final per-frame path sequence in transaction order.

Path Validation and Filtering. To restrict resource allocation to reusable paths, the tracer executes a health check during the `call_end` hook by inspecting the frame's exit status. The system discards paths resulting from VM-level exceptions, particularly out-of-gas errors, while retaining deterministic application-level terminations such as successful returns and `REVERT` operations. This distinction is critical because `REVERT` paths correspond to reproducible control-flow branches, including failed assertions or balance validations, which exhibit high reusability across transactions. Conversely, VM-level exceptions are non-deterministic and may manifest at any instruction depending on the gas limit. Tracking every potential out-of-gas point for a sequence of n opcodes would generate $O(n)$ distinct failure paths, leading to cache fragmentation without benefiting deterministic execution. Consequently, the tracer formats only deterministically terminated paths into PathLog entries for the SSA Optimizer.

3.3.2 SSA Optimizer. A pure-function component transforms a raw PathLog into an optimized, gas-annotated SsaGraph through a pipeline of graph construction, redundancy elimination, gas integration, and final compaction.

Graph construction. For each PathLog entry, the optimizer creates a node in the SsaGraph and connects it to its data-dependency predecessors using the D_{in} LSN list, yielding a graph representation of the linear trace.

Optimization passes. The graph then undergoes three side-effect-aware passes: constant folding, dead-code elimination, and common-subexpression elimination. First, PUSH nodes are converted into constant-table entries and their values are propagated through side-effect-free nodes; computations that become fully constant are removed and recorded as constants. Second, a backward scan from side-effecting nodes removes any node whose result is unused, iterating to a fixed point. Third, the optimizer merges redundant side-effect-free operations by assigning each one a fingerprint consisting of its opcode and input LSNs; nodes with identical fingerprints are unified and all consumers are redirected to the canonical node.

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.

3.3.3 Path Cache. A two-tier architecture separates prediction logic from canonical storage to efficiently support both probabilistic Online lookups and deterministic Replay retrieval.

Tiered Architecture. The Path Mapping Layer (PML) operates as a frequency-based prediction index for Online mode. It maintains a dual-index structure for each CallSig comprising a frequency map M_{freq} for $O(1)$ access and a priority queue I_{sorted} for $O(\log k)$ maximum frequency retrieval. To minimize guard validation overhead, the PML prioritizes precision by returning a prediction only if a single *PathDigest* holds the unique maximum frequency. The Graph Mapping Layer (GML) acts as the deterministic backing store mapping PathDigests to reusable SsaGraphs and DataKeys to contract-specific constant tables.

Query and Update Protocols. Query logic adapts to the operational mode. In Online mode, as detailed in Algorithm 2, the engine requests a high-confidence prediction from the PML. A successful hit yields a PathDigest that combines with the contract code hash to retrieve execution artifacts from the GML. In contrast, Replay mode bypasses the PML to perform direct lookups in the GML using the TxPlan.

Cache updates occur through two mechanisms formalized in Algorithm 3. First, the SSA Optimizer populates the GML and initializes the PML entry upon generating a new graph. Second, successful online executions trigger a feedback loop where the engine signals the PML to increment the path's access frequency. This mechanism adapts predictions to evolving traffic patterns. Fine-grained read-write locks manage concurrency by permitting parallel reads for hot path prediction while bounding write contention.

Algorithm 2: Online Mode: Path Lookup

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

Persistence and Recovery. The system serializes cache state to disk via periodic checkpoints to ensure durability. A configurable pruning policy manages storage by evicting CallSigs below a frequency threshold. Upon node restart, indices are reconstructed from checkpoints in $O(N \log k)$ time to enable immediate high-accuracy prediction. The system handles paths absent from the checkpoint via lazy regeneration.

3.3.4 Helios Engine. A central orchestrator supports transaction execution in both Online and Replay mode.

It integrates with the host EVM client by replacing the per-frame execution loop while reusing the client's state and memory management. In our Revm integration, Helios inherits arena-based memory allocation, cached and prewarmed storage access, and the transaction-local journal for atomic commits. This allows the engine to focus on optimizing intra-frame computation while leaving state handling unchanged. For each frame, Helios chooses between its Traced Interpreter and Native Interpreter based on the execution mode and the Path Cache output.

Transaction-scoped execution. Helios executes at transaction granularity. If a traced execution encounters a cache miss, guard violation, or out-of-gas condition, the engine discards all partial work and restarts the transaction on the Native Interpreter. This design avoids fine-grained checkpointing or rollback and relies on the EVM's transaction-level atomicity for correctness.

Algorithm 3: Path Frequency Update (Feedback Loop)

Notation: Symbols follow Algorithm 2. Additionally, k denotes the number of distinct frequencies in I_{sorted}.
Input: σ: CallSig corresponding to the executed path
Input: ℓ: PathDigest that was successfully executed
Input: $\mathcal{P}$: Path Cache with PML
Output: Updated frequency statistics in $\mathcal{S}_\sigma$

```
// Retrieve or create PathStore for this CallSig
```
$\mathcal{S}_\sigma \leftarrow \mathcal{P}.PML.\textsc{GetOrCreate}(\sigma)$
$\textsc{AcquireWriteLock}(\mathcal{S}_\sigma)$; `// Exclusive access for update`
`// Phase 1: Get current frequency`
$f_{old} \leftarrow M_{freq}.\textsc{Get}(\ell)$ **or** 0 ; `// Default to 0 for new paths`
$f_{new} \leftarrow f_{old} + 1$; `// Increment with saturation`
`// Phase 2: Update sorted index`
if $f_{old} > 0$ **then**
  $P_{old} \leftarrow I_{sorted}[f_{old}]$; `// Get old frequency bucket`
  $P_{old}.\textsc{Remove}(\ell)$
  if $P_{old} = \emptyset$ **then**
    $I_{sorted}.\textsc{Remove}(f_{old})$; `// Clean empty bucket`
`// Phase 3: Update sorted index`
$P_{new} \leftarrow I_{sorted}.\textsc{Entry}(f_{new}).\textsc{OrInsertEmpty}()$
$P_{new}.\textsc{Insert}(\ell)$
`// Phase 4: Update frequency map`
$M_{freq}[\ell] \leftarrow f_{new}$
$\textsc{ReleaseWriteLock}(\mathcal{S}_\sigma)$

Traced Interpreter. When the Path Cache returns a valid SsaGraph, the engine dispatches execution to the Traced Interpreter, whose loop is given in Algorithm 4. It differs from a standard EVM interpreter in three ways.

First, it replaces the EVM stack with a register-like array indexed by LSN. Each SsaGraph node writes its result to its assigned slot, and consumers read operands directly from this array, eliminating DUP/SWAP and other stack manipulation overhead.

Second, in Online mode, it enforces speculative control-flow guards. Before executing JUMP or JUMPI, the interpreter computes the runtime target and checks it against the cached target in the SsaGraph. Any mismatch triggers an immediate transaction-level fallback. In Replay mode, all control-flow targets are fixed by the TxPlan, so these guards are never violated by construction.

Third, it uses chunked gas accounting for static-cost instructions. Instructions accumulate static cost within their GasChunk, and the interpreter deducts the aggregated amount at delimiter nodes in a single operation. Instructions with dynamic gas cost perform individual gas calculations and deductions, preserving exact gas semantics while reducing the number of checks on the hot path.

Algorithm 4: Traced Interpreter Execution

Input: $\mathcal{G} = (V, E)$: Optimized SSA graph
Input: $C : \mathbb{N} \to \mathbb{U}_{256}$: Constant value table
Input: Γ: 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 $\langle \Gamma, g_{lim} - g_{rem} \rangle$

3.4 Security Considerations

Beyond the JIT Bomb resistance established through the path-driven paradigm in §2, 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 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

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×

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.

4 EVALUATION

We evaluate Helios under microbenchmarks and Ethereum mainnet workloads to answer three questions: **RQ1** (Optimization Overhead): What are the time and space costs of Helios's tracing and optimization? **RQ2** (Performance Gains): How much speedup does Helios achieve over modern EVM interpreters and existing optimizations? **RQ3** (System Applicability): How well do Replay and Online modes serve their respective deployment scenarios?

4.1 Experimental Setup

All experiments run on an AWS r7i.2xlarge instance with 8 vCPUs and 64 GB memory. We compare Helios against four baselines representing the state-of-the-art. Geth v1.9.9 [10] serves as the representative Go-based client, while Revm v22.0.1 [1] represents high-performance Rust interpreters. We also evaluate Forerunne [14], applying its full-context tracing to both Geth and Revm. Finally, we include Revmc v0.1.0 [1], a JIT compiler that translates EVM bytecode to Rust. Revmc bundles its own Revm version, which may differ from our standalone Revm baseline.

For microbenchmarks we use three DeFi workloads of increasing complexity: *ERC20-Transfer*, *Uniswap-V2-Swap-1hop*, and *Uniswap-V2-Swap-4hop*. Each transaction is executed 100 times with warm caches and we report medians. For mainnet evaluation we replay 5,000 consecutive Ethereum blocks (#19,476,587–#19,481,586), containing 921,786 transactions, of which 567,372 are contract invocations. Pure ETH transfers are excluded because they do not involve EVM execution.

4.2 Microbenchmark Performance

4.2.1 Optimization Overhead (RQ1). Table 2 summarizes tracing latency and artifact size compared to Forerunner. For ERC20-Transfer, Helios records a path in 23.2 μs, yielding a 12.5× reduction over

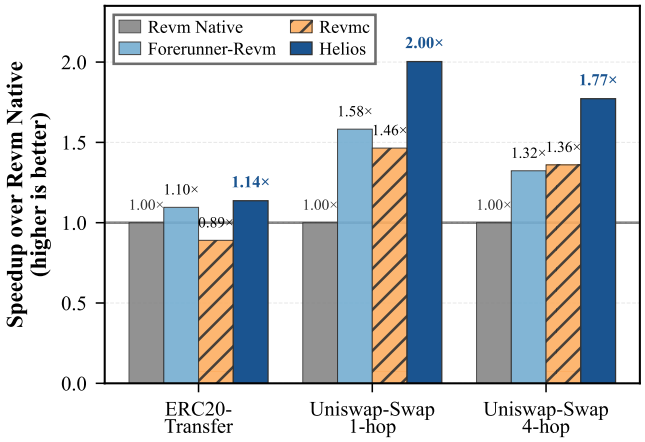

Figure 4: Execution speedup over Revm 22.0.1 baseline. Higher values indicate greater performance improvement.

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

System	ERC20 (μs)	1hop (μs)	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

Forerunner-Geth and comparable latency to our Forerunner-Revm reimplementation. As contract complexity increases, tracing cost becomes dominated by the number of executed instructions, and Helios still achieves 1.4–2.4× lower tracing time than Forerunner-Geth and about 1.2× lower than Forerunner-Revm.

The storage savings are more pronounced. For the three benchmarks, Helios reduces artifact size by 82.7×, 13.0×, and 16.2× compared to Forerunner-Geth, and by 9.4–16.4× compared to Forerunner-Revm. The resulting artifacts (4.8–122.9 KB) are small enough to be cached in memory, making online management practical.

4.2.2 Execution Speedup (RQ2). Table 3 compares Geth-based and Revm-based systems. Forerunner-Geth accelerates Geth Native by up to 5.8× but still remains substantially slower than unoptimized Revm on simple workloads, highlighting that optimization benefits are bounded by the efficiency of the underlying interpreter. We therefore focus on Revm-based systems when comparing optimization strategies.

Figure 4 reports speedups over Revm Native. Helios achieves 1.14×, 2.00×, and 1.77× speedup on ERC20-Transfer, Uniswap-1hop, and Uniswap-4hop respectively. Gains are largest on medium and complex swaps, where repetitive arithmetic and control-flow patterns allow the SSA optimizer to eliminate redundant work.

Against Forerunner-Revm, Helios improves performance by 4–34% across benchmarks. On Uniswap-1hop, Forerunner-Revm achieves 1.58× speedup, whereas Helios reaches 2.00×, reducing execution time from 39.2 μs to 31.0 μs. The advantage comes from lightweight artifacts that are faster to load and execute.

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×**
Predicted speedup	*5.66×*	*4.28×*	*4.33×*

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

Table 5: Micro-architectural latency breakdown of a hash-intensive workload (ns per iteration).

Component	Native EVM	Helios	Reduction
Heavy Ops (Keccak)	314.79	312.33	0.8%
Light Ops	46.49	45.96	1.1%
System Overhead	134.35	93.65	**30.3%**

Revmc exhibits mixed performance. While it is 11% slower than Revm Native on ERC20-Transfer, it achieves speedups of 1.46× and 1.36× on the two Uniswap workloads, respectively. JIT compilation amortizes better on longer paths, but still underperforms Helios. Interpreter-level SSA specialization thus offers competitive speedups without the runtime costs of code generation.

4.2.3 Opcode Reduction vs. Speedup (RQ2). Table 4 indicates that SSA optimization eliminates 76% to 82% of dynamic opcodes across all benchmarks, where constant folding accounts for approximately 98% of the reduction. However, the resulting speedups of 1.14× to 2.00× do not scale linearly with this reduction in instruction count.

We analyze this discrepancy in Table 5 by decomposing the micro-architectural latency of a hash-intensive workload. The breakdown reveals that computationally expensive operations such as `KECCAK256` dominate execution and account for over 60% of the wall-clock latency. Since Helios delegates these operations to the native host to ensure safety, their fixed costs limit the theoretical maximum speedup. Within the optimizable scope, Helios reduces interpretation overhead including stack manipulation and gas metering by 30.5%, which decreases the per-iteration latency from 134ns to 93ns. Consequently, while the register-based model streamlines execution logic, the overall performance gains remain bounded by the inherent computational intensity of cryptographic.

4.3 Mainnet Workload Analysis

4.3.1 Storage Overhead and Coverage (RQ1). We next examine the cost of storing optimization artifacts on real workloads. Caching

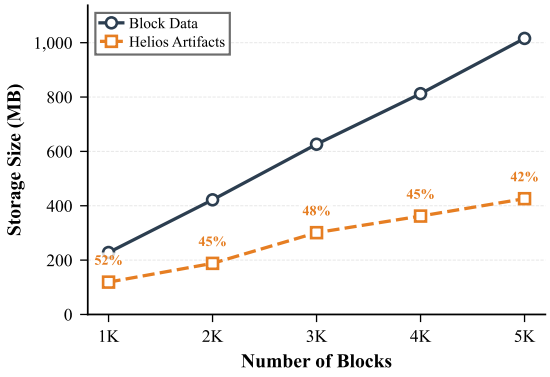

Figure 5: Storage growth for block data versus Helios optimization artifacts. Overhead decreases from 52.2% (1,000 blocks) to 41.9% (5,000 blocks) due to path convergence, where new blocks increasingly reuse existing cached artifacts.

Table 6: 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%

all unique paths observed in 5,000 blocks yields 426 MB of artifacts, corresponding to 41.9% overhead relative to raw block data (Figure 5). Overhead grows sub-linearly: the first 1,000 blocks incur 52.2% overhead, which drops as later blocks increasingly reuse existing paths.

To exploit path locality, we apply frequency-based filtering and keep only paths that execute at least f times during the warmup period (Table 6). A threshold of $f \geq 10$ reduces storage to 50 MB (4.9% overhead) while still covering 96.5% of contract executions and achieving a 58.0% Top-1 prediction hit rate. More aggressive thresholds further shrink storage but lose coverage, so we use $f \geq 10$ for Online deployment.

4.3.2 End-to-End Speedup (RQ2 & RQ3). Figure 6 presents the block-level speedup distribution across all three deployment configurations. We summarize the key observations below.

Replay Mode. Replay mode assumes a precomputed transaction plan and represents the best-case scenario for archive nodes. It achieves a median speedup of 6.60× over Revm Native, with the 75th and 90th percentiles reaching 13.88× and 21.43×, respectively. Only 5.4% of blocks exhibit slowdown. The distribution skews toward high speedups (10–20× and ≥20× bins account for 37.6% of blocks), confirming that deterministic path reuse enables substantial acceleration for historical block re-execution.

Online Mode (Full Cache). Without frequency filtering, Online mode must predict execution paths using only the current CallSig. It reaches a median speedup of 4.56×, with 75th and 90th percentiles at

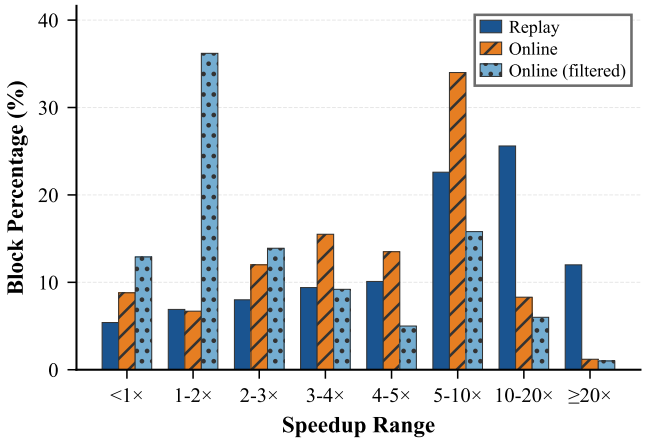

Figure 6: Block-level speedup distribution over Revm Native across three deployment modes. Replay assumes known transaction plans; Online predicts paths at runtime. Filtering ($f \geq 10$) trades median speedup for 8× storage reduction.

7.12× and 9.85×. The distribution concentrates in the 3–10× range (63.0% of blocks), reflecting the overhead of runtime prediction and occasional mispredictions. Still, 8.8% of blocks are slower than baseline—higher than Replay mode due to cache lookup costs and fallback penalties.

Online Mode (Filtered, $f \geq 10$). Frequency-based filtering reduces storage to 50 MB but shifts the distribution leftward. The median speedup drops to 2.05×, and the 1–2× bin now dominates (36.2% of blocks). However, the 90th percentile remains high at 9.01×, indicating that hot paths still benefit substantially. This trade-off suits latency-sensitive validators who prioritize storage efficiency over median-case acceleration.

Cross-Mode Comparison. The three distributions reveal a clear hierarchy: Replay > Online (full) > Online (filtered). The gap between Replay and Online stems from three factors: (i) Top-1 prediction covers only 58.0% of executions under filtering; (ii) transaction-level rollback discards partially optimized work on mispredictions; and (iii) prediction overhead becomes visible when optimized paths execute in microseconds. We discuss potential extensions in §5.

5 DISCUSSION

This section interprets the performance results, analyzes the limitations of the current design, and outlines future research directions.

5.1 Interpreting the Speedup

Helios achieves a median speedup of 6.60× in Replay Mode. This performance improvement results from the execution of optimized SSA graphs, which eliminates redundant stack operations and simplifies gas accounting for static instructions.

However, the evaluation reveals a disparity between the opcode reduction rate and the actual execution speedup. While SSA optimization removes approximately 80% of instructions, the microbenchmark speedups range from 1.14× to 2.00×. This discrepancy indicates that the overhead of the Traced Interpreter limits the potential performance gains. The management of execution metadata and the interpretation of the graph structure introduce costs that are comparable to the savings from instruction elimination. Furthermore, unoptimized heavy instructions continue to dominate the execution time in certain workloads.

A promising avenue for future work is implementing JIT or Ahead-Of-Time (AOT) compilation to eliminate interpretation overhead and fully leverage the instruction reduction achieved by our SSA optimizer. The register-based design of the SsaGraph maps naturally to modern CPU architectures, facilitating this transition. Furthermore, unlike traditional bytecode JITs that face "JIT bomb" risks, the acyclic and strictly bounded nature of the SsaGraph offers a unique opportunity for safe compilation, avoiding complexity explosion attacks.

5.2 Online Mode Challenges

Online Mode performance decreases when frequency-based filtering is applied. Our coverage table analysis reveals a gap between the Top-1 prediction coverage and actual execution coverage. This suggests that the current simplicity-first strategy does not fully exploit the optimized paths.

We explored alternative strategies to address this limitation, including a race-parallel execution model. In this approach, the system caches the Top-K paths and executes them concurrently, committing the result of the first successful path or falling back to native execution if all fail. However, microbenchmarks showed that the overhead of the parallel framework outweighed the benefits. Native execution completed in 60 μs, whereas the race-parallel implementation increased latency to 70 μs, compared to 30 μs for the baseline optimized execution in the Uniswap-1hop case. Efficiently leveraging multi-path caching remains an open problem.

Another factor is the transaction-scoped fallback mechanism. Currently, a single frame prediction failure triggers a rollback of the entire transaction to native EVM, underutilizing successful frame-level predictions. A potential solution is frame-level fallback, where only the failed frame reverts to native execution while subsequent frames continue using cached paths. However, this introduces a new attack vector: adversaries could craft malicious transactions that deliberately trigger frequent engine switching to degrade performance—we term this "de-optimization bombs." To defend against such attacks, a circuit breaker could limit the number of fallbacks per transaction. The system could also explore chained predictions, where one optimized frame's output directly triggers speculative execution of the next. These strategies present trade-offs between simplicity and coverage that merit further study.

5.3 System Scalability

Beyond execution logic, system scalability presents further optimization opportunities. While our initial exploration of instruction-level parallelism (ILP) [31] did not yield satisfactory results due to synchronization overhead, parallel execution potential remains. The current dependency graph modeling could benefit from advanced ILP techniques in compiler theory. Future research could investigate sophisticated scheduling algorithms or hardware-assisted primitives to unlock the parallelism inherent in SsaGraph.

Finally, the current Path Cache implementation prioritizes low storage overhead with a simple persistence and pruning policy. As the system scales to handle larger state histories, more mature caching strategies will be required. Future work could explore tiered storage architectures that balance hit rate, retrieval latency, and storage cost, potentially leveraging external high-performance key-value stores for long-term artifact persistence.

6 RELATED WORK

This work focuses on accelerating Ethereum Virtual Machine (EVM) execution, specifically targeting modern, high-performance interpreters such as Revm. Research in this domain can be categorized into smart contract optimization toolchains, path-driven speculative execution systems, and concurrent execution architectures.

6.1 Smart Contract Optimization

Optimization in the smart contract landscape spans from static source analysis to bytecode rewriting and compilation.

Program Analysis and Optimization. Traditional Solidity toolchains provide static analysis capabilities that serve as a foundation for downstream optimization. Tools like Slither [20] generate intermediate representations (SlithIR) and control flow graphs (CFG) to detect vulnerabilities. Rattle [8] lifts EVM bytecode into a SSA form to recover high-level control flow. While Rattle and Helios both leverage SSA to eliminate redundant stack operations, their objectives differ fundamentally. Rattle employs SSA as an intermediate representation for static analysis and decompilation, prioritizing readability without propagating changes back to the executable bytecode. In contrast, Helios utilizes SSA as a dynamic, executable format (`SsaGraph`) specifically designed for the runtime engine. The Helios optimizer operates on frame-level execution paths rather than static bytecode, generating artifacts for immediate execution acceleration via a specialized interpreter rather than for analysis.

Superoptimization. Superoptimization frameworks [12, 13, 29] aim to identify the optimal instruction sequence functionally equivalent to a target sequence but with minimal gas cost. While theoretically capable of producing maximally efficient code, these approaches rely on SMT solvers [16] to verify equivalence. The operational semantics of the EVM—specifically the modeling of memory expansion, storage, and dynamic gas costs—require complex logic encodings that heavily tax SMT solvers. Consequently, the search space for equivalent programs explodes exponentially with instruction count, often necessitating strict timeouts or restricting optimization to basic blocks without memory side-effects. These constraints make superoptimization difficult to apply dynamically at runtime.

JIT and AOT Compilation. Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilers attempt to translate EVM bytecode into native machine code. Projects like Revmc [11] generate Rust code from EVM bytecode to bypass the interpreter loop. However, for the microsecond-scale transactions typical of the EVM, the overhead of runtime code generation often outweighs the execution speedup. Helios mitigates this overhead by avoiding full native compilation. Instead, it transforms execution paths into an abstract graph representation replayed by a specialized Traced Interpreter. This approach maintains the micro-architectural optimizations of the underlying host interpreter while reducing the instruction dispatch count.

6.2 Path-driven Speculative Execution

As a path-driven execution engine, Helios is directly comparable to systems like Forerunner [14] and Seer [35].

Comparison with Existing Systems. Forerunner applies speculative execution with CD-Equiv constraints, generating "Accelerated Programs" (APs) from transaction history. Seer employs fine-grained branch prediction and snapshots to manage state dependencies. However, these systems face limitations on modern interpreters due to their tracing strategy. Forerunner relies on full-context tracing, capturing stack frames, memory snapshots, and contract state. On optimized engines like Revm, the I/O overhead of loading large artifacts often exceeds transaction execution time. Helios employs **lightweight asynchronous tracing** to address this bottleneck. By recording only stack operations, Helios achieves 9.4–16.4× compression over full-context approaches, making online artifact management practical.

Granularity and Gas Semantics. Existing systems typically utilize transaction-level caching, which limits reuse to identical transaction sequences. Helios introduces **frame-level caching**, exploiting the observation that individual contract call frames exhibit high path locality even when parent transactions differ. Furthermore, prior works do not explicitly address the preservation of gas semantics during optimization. Helios guarantees gas equivalence by construction: it restricts SSA optimization to static-cost instructions while delegating all dynamic-cost operations (e.g., memory expansion, storage access) to the native EVM, ensuring the economic model remains undisturbed.

6.3 Concurrent and Parallel Execution

Various concurrent execution architectures address the sequential bottleneck of the EVM.

Operation-Level Concurrency. ParallelEVM [28] enables concurrent execution at the operation level by dynamically generating an SSA Operation Log to track data dependencies. This mechanism allows selective re-execution of conflicting operations rather than aborting entire transactions. While both ParallelEVM and Helios leverage SSA, their objectives differ fundamentally: ParallelEVM employs SSA for concurrency control and conflict resolution, whereas Helios utilizes it for single-thread execution path optimization. Furthermore, ParallelEVM relies on synchronous runtime tracing. Although the reported log generation overhead is approximately 4.5%, Helios posits that on high-performance interpreters like Revm, any synchronous tracing on the critical path introduces non-negligible overhead, limiting net acceleration.

Parallel Architectures. Other approaches focus on architectural innovations. PaVM [19] supports both intra-contract and inter-contract parallelism through a specialized runtime system, though it does not explicitly address parallel determinism. Block-STM [24] implements optimistic concurrency control with a collaborative scheduler to execute transaction sets in parallel. Hardware-centric solutions such as MTPU [30] adopt algorithm-architecture co-design, utilizing spatial-temporal scheduling to exploit parallelism.

Orthogonality Analysis. Helios is orthogonal to these frameworks. Its core contribution is a lightweight, high-throughput path execution engine that optimizes the fundamental unit of computation—the sequential execution of a contract path. This baseline acceleration can be integrated into parallel frameworks such as ParallelEVM, Block-STM, or PaVM to further maximize system throughput.

7 CONCLUSION

We presented Helios, a path-driven execution engine designed to accelerate transaction processing on high-performance EVM clients. We identified a performance paradox in existing optimization strategies. On modern, highly optimized interpreters, the overhead of detailed tracing and artifact management often negates the benefits of optimization. Helios addresses this challenge through a novel architecture that combines lightweight asynchronous tracing with frame-level caching. By restricting optimization to static-cost instructions, Helios achieves gas-semantic equivalence by construction and eliminates the economic risks associated with aggressive JIT compilation.

Our evaluation on Ethereum mainnet workloads demonstrates the efficacy of this approach. In Replay Mode, Helios achieves a median speedup of 6.60× over the baseline Revm interpreter, validating its potential for accelerating archive node synchronization and historical data analysis. In Online Mode, Helios effectively accelerates hot execution paths, leveraging the strong path locality inherent in smart contract execution.

Helios establishes a new design point for blockchain execution engines, prioritizing safety, correctness, and architectural simplicity alongside raw performance. Future work will explore integrating JIT compilation to further reduce interpretation overhead for optimized paths and refining the speculative execution model with frame-level fallback mechanisms to maximize coverage. By decoupling optimization from the critical path, Helios provides a scalable foundation for the next generation of EVM infrastructure.

REFERENCES

[1] [n.d.]. bluealloy/revm: Rust implementation of the Ethereum Virtual Machine. https://github.com/bluealloy/revm

[2] [n.d.]. Introduction | Monad Developer Documentation. https://monad-docs-lb3l7tky7-monad-xyz.vercel.app/

[3] [n.d.]. ipsilon/evmone: Fast Ethereum Virtual Machine implementation. https://github.com/ipsilon/evmone

[4] [n.d.]. Overview | Uniswap. https://docs.uniswap.org/contracts/v2/overview

[5] [n.d.]. A Technical Deep-Dive on the JIT/AOT Compiler for revm of BNB Chain. https://www.bnbchain.org/en/blog/a-technical-deep-dive-on-the-jit-aot-compiler-for-revm-of-bnb-chain

[6] 2020. *Binance Smart Chain: A Parallel Binance Chain to Enable Smart Contracts.* Whitepaper. BNB Chain Community. https://dex-bin.bnbstatic.com/static/Whitepaper_%20Binance%20Smart%20Chain.pdf Accessed: 2025-11-24.

[7] 2021. *Polygon: Ethereum's Internet of Blockchains.* Whitepaper. Polygon Labs. https://polygon.technology/papers/pol-whitepaper Accessed: 2025-11-24.

[8] 2025. crytic/rattle. https://github.com/crytic/rattle original-date: 2018-03-06T20:48:37Z.

[9] 2025. ethereum/evmjit. https://github.com/ethereum/evmjit original-date: 2014-12-01T16:55:51Z.

[10] 2025. ethereum/go-ethereum. https://github.com/ethereum/go-ethereum original-date: 2013-12-26T13:05:46Z.

[11] 2025. paradigmxyz/revmc. https://github.com/paradigmxyz/revmc original-date: 2024-03-10T16:27:36Z.

[12] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria Anna Schett. 2022. Super-optimization of Smart Contracts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31 (2022), 1 – 29. https://api.semanticscholar.org/CorpusID:248325349

[13] Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria Anna Schett. 2020. Synthesis of Super-Optimized Smart Contracts Using Max-SMT. *Computer Aided Verification* 12224 (2020), 177 – 200. https://api.semanticscholar.org/CorpusID:219320104

[14] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based speculative transaction execution for ethereum. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 570–587.

[15] Coinbase. 2023. Base: An Ethereum L2 Network. https://docs.base.org/. Accessed: 2025-11-24.

[16] Leonardo de Moura and Nikolaj Bjørner. 2009. Satisfiability Modulo Theories: An Appetizer. In *Formal Methods: Foundations and Applications (SBMF 2009) (Lecture Notes in Computer Science)*, Vol. 5902. 23–36.

[17] Ethereum Foundation. 2025. Ethereum Archive Node. https://ethereum.org/developers/docs/nodes-and-clients/archive-nodes/. Page last updated: October 21, 2025.

[18] Ethereum Foundation. 2025. Nodes and Clients. https://ethereum.org/developers/docs/nodes-and-clients/. Page last updated: October 22, 2025.

[19] Yaozheng Fang, Zhiyuan Zhou, Surong Dai, Jinni Yang, Hui Zhang, and Ye Lu. 2024. PaVM: A Parallel Virtual Machine for Smart Contract Execution and Validation. *IEEE Transactions on Parallel and Distributed Systems* 35 (2024), 186–202. https://api.semanticscholar.org/CorpusID:265341872

[20] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)* (2019), 8–15. https://api.semanticscholar.org/CorpusID:85442214

[21] Hang Feng, Yufeng Hu, Yinghan Kou, Runhuai Li, Jianfeng Zhu, Lei Wu, and Yajin Zhou. 2024. SlimArchive: A Lightweight Architecture for Ethereum Archive Nodes. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, USA, 1257–1271. https://www.usenix.org/conference/atc24/presentation/feng-hang

[22] G. Fowler, L. C. Noll, and K.-P. Vo. 1991. Fowler-Noll-Vo Hash Algorithm. http://www.isthe.com/chongo/tech/comp/fnv/ Unpublished reviewer comments.

[23] G. Fowler, L. C. Noll, K.-P. Vo, and D. Eastlake. 2019. The FNV Non-Cryptographic Hash Algorithm. https://www.ietf.org/archive/id/draft-eastlake-fnv-17.html Internet Draft, IETF.

[24] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. 2022. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (2022). https://api.semanticscholar.org/CorpusID:247447566

[25] Go Ethereum Team. 2025. Sync Modes. https://geth.ethereum.org/docs/fundamentals/sync-modes. Last edited on February 18, 2025.

[26] Xiaowen Hu, Bernd Burgstaller, and Bernhard Scholz. 2023. EVMTracer: dynamic analysis of the parallelization and redundancy potential in the ethereum virtual machine. *IEEE Access* 11 (2023), 47159–47178.

[27] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, Private Smart Contracts. In *Proceedings of the 27th USENIX Security Symposium*. 1353–1370.

[28] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. 2025. ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems*. 211–225.

[29] Julian Nagele and Maria Anna Schett. 2020. Blockchain Superoptimizer. *ArXiv* abs/2005.05912 (2020). https://api.semanticscholar.org/CorpusID:218596321

[30] Rui Pan, Chubo Liu, Guoqing Xiao, Mingxing Duan, Keqin Li, and Kenli Li. 2023. An Algorithm and Architecture Co-design for Accelerating Smart Contracts in Blockchain. *Proceedings of the 50th Annual International Symposium on Computer Architecture* (2023). https://api.semanticscholar.org/CorpusID:259177676

[31] B. R. Rau. 1992. Data flow and dependence analysis for instruction level parallelism. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 236–250.

[32] Ben L. Titzer. 2023. Whose Baseline Compiler is it Anyway? *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2023), 207–220. https://api.semanticscholar.org/CorpusID:258833052

[33] Fabian Vogelsteller and Vitalik Buterin. 2015. *EIP-20: ERC-20 Token Standard.* Ethereum Improvement Proposal 20. Ethereum Foundation. https://eips.ethereum.org/EIPS/eip-20 Accessed: 2025-11-24.

[34] Daniel Davis Wood. 2014. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://api.semanticscholar.org/CorpusID:4836820

[35] Shijie Zhang, Ru Cheng, Xinpeng Liu, Jiang Xiao, Hai Jin, and Bo Li. 2024. Seer: Accelerating Blockchain Transaction Execution by Fine-Grained Branch Prediction. *Proceedings of the VLDB Endowment* 18, 3 (2024), 822–835.