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

Sipeng Xie[1], Qianhong Wu[1], Wenkuan Xiao[1], Mingzhe Zhai[1], Kun Wang[1], Qiyuan Gao[1], and Bo Qin[2]
[1]Beihang University, Beijing, China [2]Renmin University of China, Beijing, China
{sipengxie, qianhong.wu, wenkuanxiao, zhaimingzhe, kun_wang, gaoqy}@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], Qianhong Wu[1], Wenkuan Xiao[1], Mingzhe Zhai[1], Kun Wang[1], Qiyuan Gao[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, 24, 31], fundamentally operate as replicated state machines. From a data management perspective, each node functions as a deterministic transaction processor handling two distinct workloads. These workloads comprise the real-time processing of incoming transactions and the re-execution of historical transactions to verify global state transitions. As throughput and contract complexity increase, execution becomes a primary scalability bottleneck for both scenarios.

Recent acceleration strategies face distinct limitations on modern execution engines [1, 3]. Just-In-Time compilation introduces security risks, such as JIT bombs, and causes gas accounting discrepancies [2, 5, 9, 11, 29]. Transaction-level speculative execution degrades performance on fast engines due to the *Performance Paradox* [14, 32]. In this phenomenon, the overhead of instrumentation, tracing, and artifact management frequently exceeds the latency of transaction execution itself. Similarly, operation-level concurrent execution incurs tracing or synchronization costs that frequently outweigh parallel gains [23, 25].

Overcoming this paradox necessitates addressing three intertwined challenges. **First**, achieving net acceleration on sub-microsecond engines requires strictly decoupling heavy tasks, such as tracing and optimization, from the critical path. Synchronous overhead during transaction execution inevitably erodes optimization gains. **Second**, existing transaction-level approaches treat optimization artifacts as ephemeral. The challenge lies in architecting a persistent storage model that maximizes artifact reuse across diverse execution contexts, thereby moving beyond the limitations of single-use traces. **Third**, aggressive optimization strategies must avoid security vulnerabilities and ensure exact gas-semantic equivalence without relying on complex runtime compensation.

To assess whether these challenges can be addressed in practice, we analyze Ethereum mainnet workloads to identify high-leverage optimization opportunities. We observe pronounced *frame-level path locality*, where a small fraction of unique paths accounts for most execution time. Furthermore, along these hot paths, expensive state-access operations are relatively rare, and most instructions behave like fixed-cost computation. These insights indicate that a path-driven accelerator can focus on reusable frame-level paths and on accelerating pure computation, while leaving dynamic operations to the native handling logic.

Guided by these observations, we propose Helios, a path-driven execution engine built on three architectural pillars. To minimize critical path overhead, Helios employs asynchronous stack-only tracing to offload analysis to background threads. These traces are transformed into graph artifacts with explicit data dependencies. These artifacts are then executed by a specialized register-based interpreter to eliminate redundant stack manipulations. To efficiently organize and utilize artifacts, Helios implements frame-level caching to enable a unified dual-mode design. This design includes a Replay Mode that maximizes throughput for historical transaction execution by reusing precomputed paths and an Online Mode that

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

leverages path locality to speculatively accelerate hot paths for incoming transactions. Finally, to ensure safety, Helios optimizes only executed healthy paths and adopts a hybrid execution model. This model bulk-deducts static gas while delegating dynamic operations to the native host, guaranteeing gas consistency by construction.

In summary, this paper makes the following contributions:

- We investigate the limitations of transaction-level speculation and operation-level concurrency on modern execution engines to formulate the *Performance Paradox*, where auxiliary overheads negate optimization gains. We further identify frame-level path locality and the separation of static and dynamic costs as key levers to resolve this paradox.
- We propose a novel architecture that decouples optimization from critical execution paths via asynchronous lightweight tracing and maximizes artifact reuse through frame-level caching. This approach ensures minimal instrumentation overhead and high cache hit rates.
- We develop the high-performance Helios Engine utilizing a register-based interpreter and bulk gas deduction to minimize execution overhead. It ensures intrinsic gas consistency through a hybrid model that strictly separates static optimizations from dynamic costs. This engine serves as the versatile runtime powering both deterministic replay and speculative execution within our unified framework.
- We implement a prototype of Helios on Revm and evaluate it using Ethereum mainnet workloads. The results demonstrate a median speedup of 6.60× over the native baseline on historical workloads while providing effective acceleration for incoming transactions.

2 BACKGROUND & MOTIVATION

2.1 The EVM Execution Model

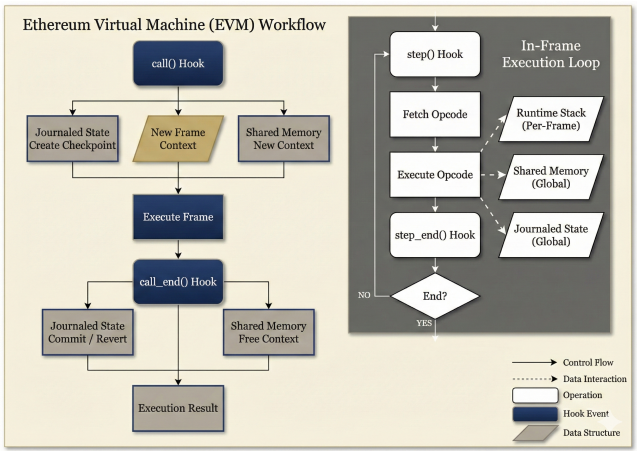

Figure 1: The EVM execution workflow. The process is partitioned into a high-level frame management lifecycle (left) and a low-level, per-opcode execution loop (right).

The EVM operates as a quasi-Turing-complete stack machine[31]. Unlike register-based architectures, the EVM performs all computations on a transient runtime stack using specific 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. Dynamic costs depend on runtime states such as memory expansion or storage access frequency. This distinction enables optimization strategies that aggregate static costs while delegating dynamic accounting to the native handling logic.

Crucially, 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. Helios leverages these standard hooks to observe execution states passively and capture data dependencies without requiring invasive modifications to the core interpreter logic.

2.2 The Performance Paradox

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

Contract-driven optimization, exemplified by Just-In-Time (JIT) compilation, faces two critical limitations in permissionless blockchains. First, it introduces an attack surface known as "JIT bombs" where attackers construct pathological code patterns, such as deeply nested conditional branches, to trigger exponential compilation complexity. This computational asymmetry allows malicious actors to exhaust validator CPU resources via low-gas transactions and constitutes a Denial-of-Service vector [2, 5, 9, 11, 29]. Second, EVM consensus rules mandate strict gas-semantic equivalence. Aggressive compiler optimizations like instruction reordering often alter the observable gas schedule, making consensus preservation difficult without complex runtime compensation. Consequently, path-driven optimization emerges as a safer paradigm by inherently bounding optimization costs to the actual execution trace.

However, existing path-driven schemes encounter a scalability barrier on modern EVM clients, a phenomenon we term the *Performance Paradox*. On slower engines like Geth [10], transaction execution dominates end-to-end latency, rendering the additional instrumentation cost negligible. Conversely, on highly optimized engines like Revm [1], the critical path shifts as execution becomes inexpensive, causing auxiliary tasks such as tracing and artifact

management to become the primary bottleneck. Table 1 demonstrates this shift using a Uniswap V2 swap [4]. A prior path-driven system [14] achieves a 5.8× speedup on Geth but only 1.6× on Revm. Two factors explain this performance degradation. First, synchronous tracing dominates the critical path. On Revm, tracing a transaction is approximately six times slower than native execution and nearly ten times slower than optimized execution, effectively negating the potential benefits. Second, artifact management incurs substantial fixed costs. Loading and parsing a 663 KB artifact to accelerate a task finishing in tens of microseconds introduces I/O latency that does not scale with the underlying execution time. These constraints necessitate two architectural shifts: replacing synchronous tracing with asynchronous optimization off the critical path and substituting large artifacts with lightweight traces to minimize data volume.

2.3 The Granularity Mismatch

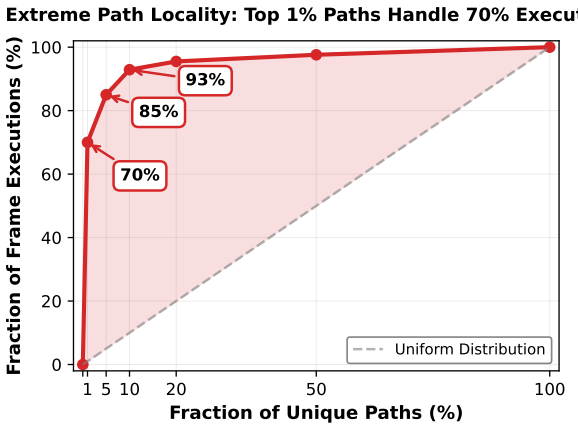

Figure 2: Path locality in EVM execution exhibits an extreme Pareto distribution.

Beyond tracing overhead, existing strategies predominantly employ transaction-level caching [14, 25, 32]. This approach treats the entire transaction execution trace as the atomic unit of optimization. It faces a combinatorial challenge because slight variations in the sequence of internal contract calls render the entire cached artifact invalid. This dependency enforces a "use-once-discard" lifecycle that limits reuse potential.

In contrast, our analysis of Ethereum mainnet workloads reveals significant redundancy at the finer frame granularity. As shown in Figure 2, the distribution of unique frame paths follows a Pareto principle where the top 1% of paths account for over 70% of total executions. This disparity indicates that while unique transaction combinations are vast, the 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 the amortization of compilation costs across thousands of invocations.

2.4 Design Implications

The preceding analysis suggests three architectural principles for a high-performance EVM accelerator. First, the distinction between static and dynamic gas costs motivates a hybrid execution model. Optimizing only static instructions while delegating dynamic operations to the native handling ensures gas-semantic equivalence by construction. Second, the Performance Paradox necessitates asynchronous lightweight tracing and optimization. Decoupling these tasks from the critical path limits I/O overhead and preserves baseline execution speed. Third, the pronounced path locality implies the need for frame-level persistent caching. This granularity enables compositional artifact reuse across diverse transactions, overcoming the limitations of transaction-level approaches. These principles collectively inform the design of Helios.

3 THE DESIGN OF HELIOS

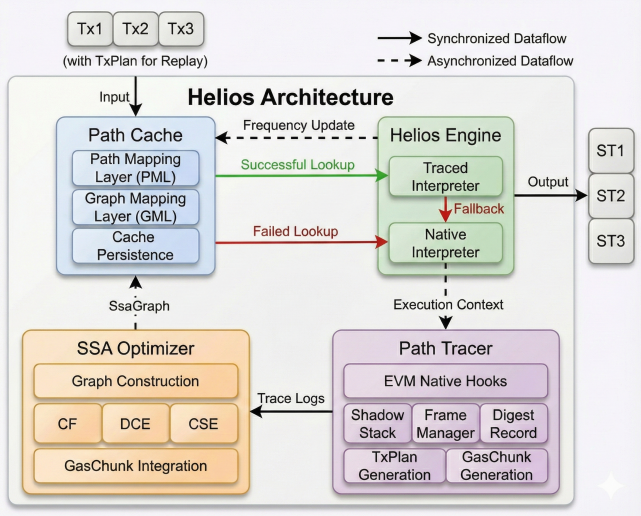

Figure 3: Overview of the Helios architecture.

3.1 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 3 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 [16], 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.

3.2 End-to-End Transaction Lifecycle

Helios operates in distinct Online and Replay modes. Online mode targets real-time processing with unknown execution paths, whereas Replay mode facilitates high-throughput historical processing using predetermined paths.

3.2.1 Online Mode: Executing New Transactions. Online mode addresses the real-time requirements of full nodes and validators handling transactions with unknown execution paths. Upon contract invocation, the Helios Engine generates a path identifier from the contract identity and the call signature to query the Path Cache. A cache hit engages the Traced Interpreter, which performs speculative execution validated by runtime control-flow guards. Conversely, a cache miss or guard failure necessitates a fallback to the Native Interpreter to ensure correctness. This fallback initiates an asynchronous optimization pipeline where the Path Tracer records the execution sequence. The SSA Optimizer subsequently transforms this linear trace into an optimized SsaGraph and associated constants, populating the Path Cache for future reuse.

3.2.2 Replay Mode: Executing Historical Transactions. Replay mode targets the high-throughput batch processing needs of archive nodes. In this deterministic environment, the Helios Engine retrieves a pre-computed transaction plan containing an ordered list of execution path identifiers. For each call frame, the engine queries the Path Cache using the specific identifier. Since the plan references only previously verified paths, cache lookups are guaranteed to succeed. The Traced Interpreter executes the retrieved SsaGraph directly, bypassing control-flow guards and the asynchronous optimization pipeline. This approach eliminates speculation overhead and maximizes throughput for historical block reprocessing.

3.3 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 [30] 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 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 per chunk, thereby reducing overhead while preserving gas-semantic equivalence.

3.4 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.

3.4.1 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. This yields PathLog entries recording each operation alongside the LSNs of its true data dependencies.

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
```

PathDigest Calculation. PathDigest is a rolling hash updated with each executed opcode using the lightweight FNV-1A algorithm [20, 21]. 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.

3.4.2 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.

3.4.3 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.

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
```

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
S_σ ← P.PML.GetOrCreate(σ)
AcquireWriteLock(S_σ) ;   // Exclusive access for update
// Phase 1: Get current frequency
f_old ← M_freq.Get(ℓ) or 0 ;  // Default to 0 for new paths
f_new ← f_old + 1 ;             // Increment with saturation
// Phase 2: Update sorted index
if f_old > 0 then
    P_old ← I_sorted[f_old] ;      // Get old frequency bucket
    P_old.Remove(ℓ)
    if P_old = ∅ then
        I_sorted.Remove(f_old) ;        // Clean empty bucket
// Phase 3: Update sorted index
P_new ← I_sorted.Entry(f_new).OrInsertEmpty()
P_new.Insert(ℓ)
// Phase 4: Update frequency map
M_freq[ℓ] ← f_new
ReleaseWriteLock(S_σ)
```

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.

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.

3.4.4 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.

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.

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 $op(v) \in \{$JUMP, JUMPI$\}$ **then**
    $pc \leftarrow$ 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 $op(v) \in \{$GAS, RETURN, STOP, REVERT, CREATE, 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`
  $\rho \leftarrow$ Exec$(op(v), \vec{v}, \Gamma)$
  `// Deduct dynamic gas costs`
  if IsDynamic$(op(v))$ **then**
    $\delta_d \leftarrow$ 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 $op(v) \in \{$RETURN, STOP, REVERT$\}$ **then**
    return $\langle \Gamma, g_{lim} - g_{rem} \rangle$
return $\langle \Gamma, g_{lim} - g_{rem} \rangle$

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.

3.5 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 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.

4 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. §4.1 describes the experimental setup, including hardware configuration, baseline systems, and workload selection. §4.2 presents microbenchmark results for three representative DeFi transactions, isolating the impact of path tracing overhead, execution speedup, and SSA optimization effectiveness. §4.3 analyzes Helios's performance on mainnet blocks, validating path locality assumptions and measuring aggregate throughput improvements.

4.1 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 reimplemented 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.

4.2 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.

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×

4.2.1 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 μs, 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.

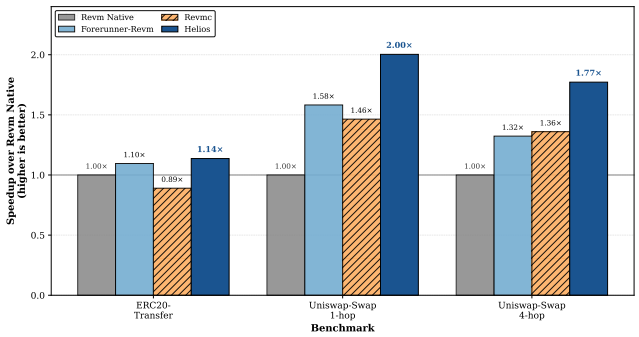

Figure 4: Execution speedup over Revm 22.0.1 baseline. Higher values indicate greater performance improvement.

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

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

4.2.2 Execution Performance. This subsection addresses **RQ2** by evaluating execution speedup on optimized paths. Figure 4 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 μs to 31.0 μs 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 μs for ERC20-Transfer remains 7.2× slower than unoptimized Revm at 4.94 μs. This disparity diminishes on complex workloads, with Forerunner-Geth achieving near-parity on Uniswap-4hop at 167.38 μs versus 172.49 μs. 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 μs to 30.96 μs.

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.

4.2.3 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.

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.

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 §2, these operations involve dynamic gas metering. We leave these operations unoptimized to avoid gas metering complications.

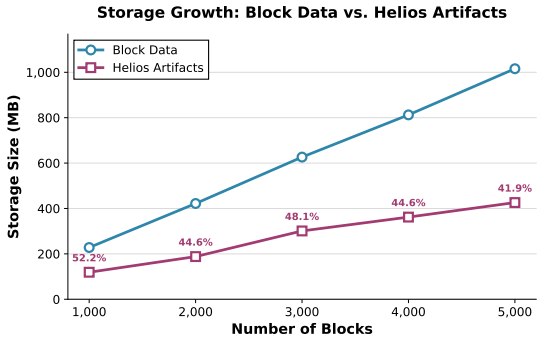

Figure 5: Storage growth for block data versus Helios optimization artifacts. Overhead decreases from 52.2% (1,000 blocks) to 41.9% (5,000 blocks) due to path convergence, where new blocks increasingly reuse existing cached artifacts.

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. §5 explores further optimization opportunities under the current design framework.

4.3 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.

4.3.1 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 5 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 52.2% overhead, decreasing to 41.9% at 5,000 blocks due to path convergence.

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%

Frequency-Based Filtering. Path locality established in §2 demonstrates that execution frequency follows a Pareto distribution. We exploit this property through frequency-based filtering, retaining only paths executed at least f times across the 5,000-block warmup period. This approach offers implementation simplicity compared to percentile-based thresholds while naturally adapting to workload characteristics. Table 5 quantifies the storage-coverage tradeoff across different frequency thresholds.

Applying frequency ≥10 filtering reduces storage to 50 MB while maintaining 96.5% execution coverage and 58.0% Top-1 hit rate. More aggressive thresholds yield diminishing returns. Frequency ≥100 achieves only 1.9% overhead but sacrifices 11% execution coverage. Frequency ≥500 becomes impractical at 69.9% coverage despite minimal 0.3% storage cost. The frequency ≥10 threshold balances storage efficiency and coverage preservation, motivating its selection for Online mode deployment. We investigate the performance implications of this filtering strategy below.

4.3.2 Replay Mode Performance. This subsection addresses **RQ2** and **RQ3** by evaluating Replay mode on 5,000 blocks to establish performance upper bounds under perfect path knowledge for archive node scenarios. This represents the oracle baseline where all execution paths are known deterministically.

Figure 6 presents the speedup distribution. Helios achieves a median speedup of 6.60×. The distribution exhibits strong tail performance with the 75th percentile at 13.88× and the 90th percentile at 21.43×. Only 5.4% of blocks experience slowdown relative to baseline execution. The distribution concentrates in the 5-10× range at 22.6% and 10-20× range at 25.6%, demonstrating consistent acceleration across diverse workloads.

Replay mode proves particularly valuable for archive nodes performing historical synchronization, where blockchain history is optimized once and replayed efficiently for subsequent queries or re-synchronizations.

4.3.3 Online Mode Effectiveness. This subsection addresses **RQ2** and **RQ3** by evaluating Online mode for live transaction execution where execution paths are unknown a priori. We evaluate performance on 1,000 blocks immediately following the 5,000-block warmup period, ensuring the test set contains fresh transactions not seen during cache construction. We evaluate performance under two configurations. Online mode without filtering caches all paths from the warmup period. Online mode with frequency ≥10 filtering

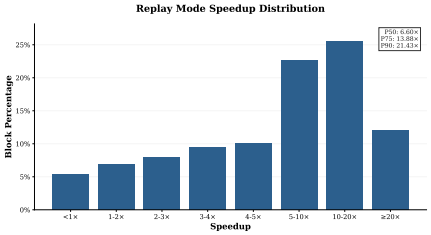

Figure 6: Replay mode speedup distribution across 5,000 mainnet blocks. 5.4% of blocks exhibit slowdown. The median speedup of 6.60× and 90th percentile of 21.43× demonstrate consistent acceleration.

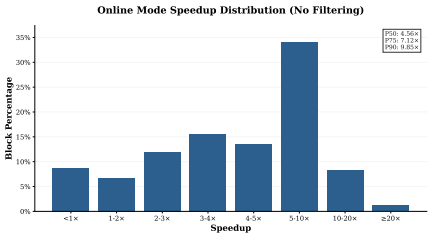

Figure 7: Online mode speedup distribution without filtering achieves median speedup of 4.56×. The distribution remains broad with 34.0% of blocks in the 5-10× range.

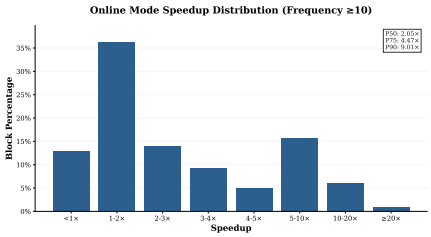

Figure 8: Online mode speedup distribution with frequency ≥10 filtering. Retaining high-frequency paths reduces median speedup to 2.05× but preserves 90th percentile at 9.01×.

retains only paths exceeding the frequency threshold, as motivated by Table 5.

Online Mode Without Filtering. Figure 7 presents the speedup distribution when caching all paths. Online mode achieves a median speedup of 4.56×, underperforming Replay mode's 6.60× by 31%. The distribution concentrates in the 5-10× range at 34.0%, with the 75th percentile at 7.12× and the 90th percentile at 9.85×. Only 8.8% of blocks exhibit slowdown relative to baseline execution.

Online Mode With Frequency Filtering. Figure 8 presents the speedup distribution under frequency ≥10 filtering. Median speedup drops to 2.05×, representing a 55% reduction relative to unfiltered Online mode. The distribution shifts toward lower speedup ranges, with 36.2% of blocks concentrated in the 1-2× range. However, tail performance remains competitive at the 90th percentile with 9.01× speedup. Performance degradation primarily affects median and lower percentiles rather than tail workloads.

Performance Degradation Analysis. Three factors contribute to Online mode's performance degradation relative to Replay mode. First, greedy Top-1 path selection achieves only 58.0% hit rate under frequency ≥10 filtering, causing 42% of transactions to fall back to native execution. Contracts with multiple high-frequency paths driven by input-dependent control flow cannot be fully covered by single-path caching. Second, Helios employs transaction-level rollback for path mispredictions. When a path divergence occurs during execution, the entire transaction rolls back to native execution rather than switching to an alternative path mid-execution. This all-or-nothing strategy amplifies the penalty for cache misses, as partially completed optimized execution provides no benefit. Third, per-transaction overhead from path prediction and cache lookup becomes proportionally significant when optimized paths execute in microseconds. These limitations suggest opportunities for multi-path caching, incremental rollback mechanisms, and adaptive path selection strategies, which we discuss 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 direction for future work is to implement Just-In-Time (JIT) or Ahead-Of-Time (AOT) compilation. Compiling the execution logic of the Traced Interpreter into native machine code would eliminate the interpretation overhead. The register-based design of the SsaGraph facilitates this transition as it maps naturally to modern CPU architectures. This approach would allow the system to fully leverage the instruction reduction achieved by the SSA optimizer.

5.2 Online Mode Challenges

The performance of Online Mode exhibits a reduction when frequency-based filtering is applied. Our analysis of the coverage table reveals a gap between the coverage achieved by the Top-1 prediction strategy and the actual execution coverage. This suggests that the current simplicity-first selection strategy does not fully utilize 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 contributing to the performance degradation is the transaction-scoped fallback mechanism. Currently, a single frame prediction failure triggers a rollback of the entire transaction to the native EVM, resulting in the underutilization of successful frame-level predictions. A potential solution is to implement frame-level fallback, where only the failed frame reverts to native execution while subsequent frames attempt to use cached paths. To mitigate the risk of frequent engine switching, which we term "de-optimization bombs," an optimization circuit breaker could limit the number of allowed fallbacks per transaction. Furthermore, the system could explore chained predictions, where the output of one optimized frame directly triggers the speculative execution of the next. These strategies represent a trade-off between architectural simplicity and execution coverage that warrants further investigation.

5.3 System Scalability

Beyond execution logic, system scalability presents additional opportunities for optimization. While our initial exploration of instruction-level parallelism (ILP) [28] did not yield satisfactory results due to synchronization overhead, the potential for parallel execution remains. The current dependency graph modeling could be enhanced by incorporating advanced ILP techniques from compiler theory. Future research could investigate more sophisticated scheduling algorithms or hardware-assisted synchronization primitives to unlock the parallelism inherent in the 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 and Compilation

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 [19] 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, 26] 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 [17] 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 [32].

Comparison with Existing Systems. Forerunner applies constraint-driven speculative execution, generating "Accelerated Programs" (APs) based on transaction history. Seer employs fine-grained branch prediction and snapshots to manage state dependencies. However, these systems face limitations when applied to modern, fast interpreters due to their tracing strategy. Forerunner relies on full-context tracing, capturing stack frames, memory snapshots, and contract state. On highly optimized engines like Revm, the I/O overhead of loading these large artifacts often exceeds the execution time of the transaction itself. Helios employs **lightweight asynchronous tracing** to address this bottleneck. By recording only stack operations and minimal data dependencies, Helios achieves compression rates of 9.4× to 16.4× compared to full-context approaches, rendering 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 [25] enables operation-level concurrency 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 [18] supports both intra-contract and inter-contract parallelism through a specialized runtime system, though it does not explicitly address parallel determinism. Block-STM [22] implements optimistic concurrency control with a collaborative scheduler to execute transaction sets in parallel. Hardware-centric solutions such as MTPU [27] 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.

6.4 Alternative Virtual Machine Environments

The emergence of WebAssembly [33] (WASM) in blockchain has led to new runtime environments like DTVM [34] and Wasmtime, which utilize JIT compilation and "hot-switching" mechanisms to balance startup performance with long-term execution efficiency. Helios adopts a similar philosophy for the EVM, using a dual-mode approach (Online vs. Replay) to adaptively balance the overhead of tracing against the benefits of optimized execution.

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, eliminating 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 with modest storage overhead, 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 Just-In-Time (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] Ronald Gary Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1991), 451 – 490. https://api.semanticscholar.org/CorpusID:13243943

[17] 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.

[18] 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

[19] 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

[20] 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.

[21] 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.

[22] 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

[23] 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.

[24] 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.

[25] 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.

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

[27] 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

[28] 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.

[29] 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

[30] 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.

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

[32] 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.

[33] Shuyu Zheng, Haoyu Wang, Lei Wu, Gang Huang, and Xuanzhe Liu. 2020. VM Matters: A Comparison of WASM VMs and EVMs in the Performance of Blockchain Smart Contracts. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 9 (2020), 1 – 24. https://api.semanticscholar.org/CorpusID:227247977

[34] Wei Zhou, Changzheng Wei, Ying Yan, Wei Tang, Zhihao Chen, Xiong Xu, Xuebing Huang, Wengang Chen, Jie Zhang, Yang Chen, Xiaofu Zheng, Hanghang Wu, Shenglong Chen, Ermei Wang, Xiang Zhou Chen, Yang Yu, Mengyu Wu, Tao Zhu, Liwei Yuan, Feng Yu, A. feng Zhang, Wei Wang, Ji Luo, Zhengyu He, and Wenbiao Zhao. 2025. DTVM: Revolutionizing Smart Contract Execution with Determinism and Compatibility. *ArXiv* abs/2504.16552 (2025). https://api.semanticscholar.org/CorpusID:277999808