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

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

ABSTRACT

Smart contract execution stands as a critical scalability bottleneck for EVM-based blockchains. Prevalent acceleration strategies, notably JIT compilation and path-driven speculative execution, face fundamental limitations. JIT approaches compromise gas accounting determinism and introduce security vulnerabilities exploitable by pathological code patterns. Conversely, path-driven methods encounter an "optimization dilemma" on hyper-optimized clients like Revm, where the overhead of tracing and artifact management negates execution gains. Furthermore, their reliance on transaction-level granularity results in ephemeral artifacts with limited reuse due to combinatorial path explosion.

We present Helios, a path-driven execution engine that overcomes these barriers through three architectural principles. First, hybrid execution restricts optimization to static-cost instructions, guaranteeing gas-semantic equivalence by construction. Second, asynchronous tracing decouples profiling from the critical path, thereby eliminating instrumentation bottlenecks. Third, frame-level caching exploits the strong path locality of individual contract calls, transforming ephemeral traces into reusable artifacts. The resulting SSA graphs are executed by a guarded register-based interpreter that ensures correctness via native fallback upon path divergence.

Evaluation on the Ethereum mainnet demonstrates a median speedup of 6.60× in Replay Mode. In Online Mode, frequency-based filtering achieves 2.05× acceleration with a negligible 4.9% storage overhead. These results establish Helios as a scalable and safe foundation for next-generation EVM infrastructure.

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

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

1 INTRODUCTION

Smart contract platforms [19, 44, 52] have extended blockchain functionality far beyond cryptocurrency transfers, enabling decentralized finance [47], non-fungible tokens [46], and on-chain governance [25]. At the core of this ecosystem lies the Ethereum Virtual Machine, or EVM, a shared execution layer that powers Ethereum mainnet, Layer-2 rollups, and EVM-compatible sidechains [1, 3, 21, 35, 48]. Together, these systems secure assets worth tens of billions of dollars and process the majority of programmable on-chain activity [8].

They typically follow a Dissemination–Consensus–Execution pipeline [20], in which transactions are broadcast through a peer-to-peer network, ordered into blocks by a consensus protocol, and executed by the EVM on every node to update local state. Because each node must independently re-execute every transaction to verify state transitions, the cost of EVM bytecode interpretation is amplified across thousands of replicas.

A node keeps pace with the chain only if it completes per-block execution within the block interval, and throughput is therefore bounded by the slower of block production and block execution [40, 51]. Consensus-layer advances have progressively shortened block intervals [23, 36, 37, 49], shifting the bottleneck toward execution. Attempts to compensate by raising gas limits or packing more transactions per block [6, 14] only intensify execution pressure, inflating per-block latency and ultimately capping achievable throughput. EVM execution has thus become the dominant scalability constraint, particularly for contract-intensive workloads.

From a data-management perspective, the EVM instantiates a deterministic transaction processor on every node that must sustain two workloads, namely real-time processing of incoming transactions and high-throughput replay of historical transactions for state verification and rollup proving. Because all nodes must agree on state transitions and gas metering underpins both economic consensus and DoS resistance, any viable acceleration strategy must improve execution speed while strictly preserving gas-semantic correctness and cross-replica determinism.

Achieving meaningful acceleration on modern, hyper-optimized execution engines [5, 12] involves overcoming three fundamental tensions. **First**, the *Compilation Limitation*, where standard compiler optimizations can alter observable execution traces, violating gas accounting rules and introducing security vulnerabilities exploitable for denial-of-service attacks [4, 9, 11, 13, 43]. **Second**, the *Optimization Dilemma* arises because instrumentation overhead on sub-microsecond engines frequently exceeds execution latency, while fine-grained parallelism incurs synchronization costs that often outweigh gains [20, 34, 38, 50]. **Third**, the *Granularity*

Mismatch, where transaction-level caching is susceptible to combinatorial path explosion, often resulting in ephemeral artifacts with limited reuse potential [20, 50].

To assess whether these barriers can be surmounted, we analyzed Ethereum mainnet workloads and derived three critical insights. **First**, fixed-cost computational instructions dominate hot paths, while dynamic state-access operations remain sparse [18, 41]. **Second**, optimization-relevant dependencies are largely confined to stack operations; lightweight stack-only tracing suffices. **Third**, frame-level paths exhibit Pareto-like locality, with a small fraction of unique paths dominating total execution time.

Guided by these insights, we present Helios, a path-driven execution accelerator. Helios is built on three architectural principles, namely *hybrid execution* that restricts optimization to static-cost instructions while delegating dynamic operations to the native engine, *asynchronous lightweight tracing* that decouples profiling from the critical path, and *frame-level caching* that exploits execution locality for high reuse. This unified architecture serves both latency-sensitive validators in Online Mode and throughput-oriented archive nodes in Replay Mode.

Realizing this architecture requires addressing two additional challenges, namely *path divergence* when runtime conditions differ from profiled traces, and *cache-flooding attacks* from adversarially generated paths. Helios mitigates these risks through *control-flow guards* that detect divergence and trigger native fallback, and *frequency-based filtering* that admits only statistically significant paths into the acceleration pipeline.

In summary, this paper makes the following contributions:

- We identify the *Optimization Dilemma* on modern execution engines and propose **asynchronous lightweight tracing** that decouples trace generation from the critical path, addressing the overhead barrier without blocking execution.
- We design **frame-level caching with frequency-based filtering** that exploits execution locality to transform ephemeral artifacts into reusable components while providing inherent resistance to path-explosion attacks.
- We develop a **guarded register-based interpreter** that accelerates execution through direct data access and bulk gas deduction, while runtime control-flow guards ensure semantic equivalence with the native engine.
- We implement Helios on Revm and demonstrate its **architectural versatility**, achieving 6.60× median speedup in Replay Mode and 2× acceleration in Online Mode for real-time validation.

Taken together, Helios advances the state of the art along four dimensions. *Safety*: hybrid execution preserves gas-semantic equivalence by construction, addressing a challenge that has limited JIT and superoptimization approaches. *Efficiency*: asynchronous tracing resolves the optimization dilemma, while the guarded register-based interpreter achieves 6.6× median speedup on the already highly optimized Revm. *Robustness*: control-flow guards and frequency filtering protect against path divergence and cache-flooding attacks. *Versatility*: a dual-mode architecture unifies the acceleration landscape, serving both latency-sensitive validators and throughput-oriented archive nodes within a single framework.

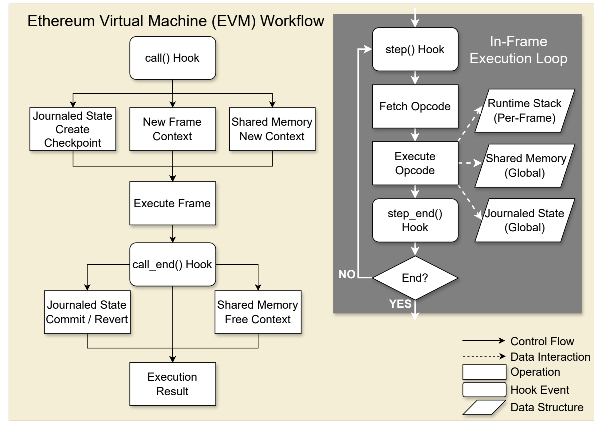

Figure 1: The EVM execution workflow. The process is partitioned into a high-level frame management lifecycle (left) and a low-level, per-opcode execution loop (right).

2 BACKGROUND & MOTIVATION

2.1 EVM Architecture and Workloads

The Ethereum Virtual Machine operates as a quasi-Turing-complete stack machine [48]. 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.

Given this execution model, the performance characteristics of EVM clients vary depending on their operational role. The Ethereum network depends on node archetypes that exhibit divergent operational characteristics [27]. Full nodes operate within fixed block intervals and prioritize the low-latency processing of unpredictable transactions to maintain consensus stability. Archive

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×	–

nodes maintain comprehensive ledger history and emphasize execution throughput to facilitate bulk re-execution of historical states [26, 30, 33].

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. Existing acceleration strategies, however, tend to optimize for one mode at the expense of the other. Systems designed for replay prioritize throughput but lack the responsiveness required for online validation [30]. Conversely, systems targeting online execution prioritize latency but produce ephemeral artifacts with limited reuse potential for historical replay [20, 50]. A unified acceleration framework must bridge this divide, achieving both low-latency responsiveness and high-throughput batch processing within a single architectural paradigm.

2.2 Challenges of EVM Acceleration

Accelerating EVM execution while preserving semantic correctness presents three interconnected challenges: the *Compilation Limitation*, the *Optimization Dilemma*, and the *Granularity Mismatch*. We examine each in turn.

2.2.1 The Compilation Limitation. Contract-level optimization, exemplified by JIT compilation, faces two critical limitations in permissionless blockchain environments. **First**, JIT compilation can introduce 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 [4, 9, 11, 13, 43]. **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 [11, 13].

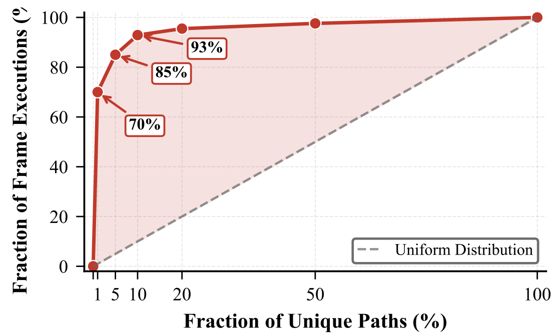

Figure 2: Path locality in EVM execution follows an extreme Pareto distribution: the top 1% of unique paths account for 70% of frame executions (5,000 mainnet blocks). The shaded region shows deviation from uniform distribution.

These constraints suggest that contract-level compilation faces significant challenges in permissionless execution environments. Path-driven optimization offers a potential alternative by restricting acceleration to actually executed paths, thereby bounding optimization costs to the runtime trace and inherently avoiding dead code and unreachable branches. However, path-driven approaches introduce a distinct correctness challenge [20, 38, 50]. Cached paths are derived from historical traces, but blockchain state evolves continuously as new transactions modify storage. When runtime conditions diverge from the profiled trace, executing a cached path without verification risks producing incorrect state transitions. This path-divergence problem demands careful mitigation in any path-driven design. Beyond correctness, existing path-driven approaches encounter a second barrier on modern execution engines.

2.2.2 The Optimization Dilemma. Existing path-driven schemes face a phenomenon we term the *Optimization Dilemma*. On slower engines such as Geth [10], transaction execution dominates end-to-end latency, rendering additional instrumentation costs negligible. A prior path-driven system [20] achieves a 5.8× speedup on Geth. However, on highly optimized engines such as Revm [5], execution becomes inexpensive and auxiliary overhead emerges as the primary bottleneck. The same system achieves a more modest 1.6× on Revm. Table 1 quantifies this shift using a representative Uniswap V2 swap transaction [2].

Three categories of auxiliary overhead contribute to this dilemma. **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. Synchronous instrumentation on the critical path tends to negate 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 yields limited benefits. We implemented a dependency-graph-driven parallel executor on Revm to evaluate instruction-level parallelism. As shown in Table 1, this approach achieves approximately 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 [34, 38]. Coordination overhead typically exceeds parallel gains at this granularity.

These observations collectively suggest that synchronous tracing, heavyweight artifacts, and fine-grained parallelism present significant challenges for achieving meaningful speedup on high-performance execution engines. Overcoming this dilemma requires fundamentally rethinking where and how optimization occurs relative to the critical execution path.

2.2.3 The Granularity Mismatch. Beyond auxiliary overhead, existing path-driven strategies employ transaction-level caching [20, 38, 50], treating the entire execution trace as the atomic optimization unit. This granularity creates a combinatorial challenge where slight variations in call sequences invalidate cached artifacts, often resulting in a use-once-discard lifecycle that limits reuse.

Our analysis of Ethereum mainnet workloads reveals significant redundancy at finer granularity. As shown in Figure 2, frame-level paths follow a Pareto distribution where the top 1% accounts for over 70% of execution time. While transaction combinations are vast, individual contract calls function as repetitive building blocks, and shifting the optimization unit to frames amortizes costs across thousands of invocations. However, finer granularity alone does not eliminate cache-flooding risk: adversaries can craft contracts generating numerous unique paths, each triggering insertion without reuse. This attack surface demands robust defenses against adversarial path generation.

2.3 Our Approach

The preceding analysis motivates Helios, a path-driven execution engine designed for safety, efficiency, robustness, and versatility. Helios adopts three architectural principles that directly address the challenges above.

To address the *Compilation Limitation*, Helios abandons full JIT compilation in favor of a hybrid execution model. The engine compiles only static-cost instructions into a register-based intermediate representation for acceleration, while delegating dynamic-cost operations to the native interpreter. This separation preserves gas-semantic equivalence by construction. To mitigate the risk of path divergence in a mutable blockchain environment, Helios embeds lightweight *control-flow guards* within optimized paths. These guards verify jump targets against the current execution context and trigger immediate fallback to native execution upon detecting any deviation.

To resolve the *Optimization Dilemma*, Helios decouples instrumentation from the critical path through an asynchronous tracing pipeline that generates optimization artifacts in the background without blocking transaction processing. Leveraging the insight that relevant dependencies are confined to the stack, Helios employs a lightweight tracer that captures necessary data flows without the prohibitive overhead of full memory or storage snapshots. The resulting hot paths are transformed into a register-based intermediate representation that achieves acceleration through direct data access and bulk gas deduction across static instruction sequences.

To overcome the *Granularity Mismatch*, Helios shifts the atomic unit of caching from transactions to call frames. This granularity exploits the high path locality of individual contract calls, converting ephemeral traces into reusable components. To defend against cache-flooding attacks, the system integrates *frequency-based filtering* that admits only statistically significant paths into the acceleration pipeline. This frame-level architecture also unifies the acceleration paradigm, allowing the same guarded artifacts to serve as predictive accelerators for latency-sensitive validators in Online Mode and as deterministic processors for throughput-oriented historical replay in Replay Mode.

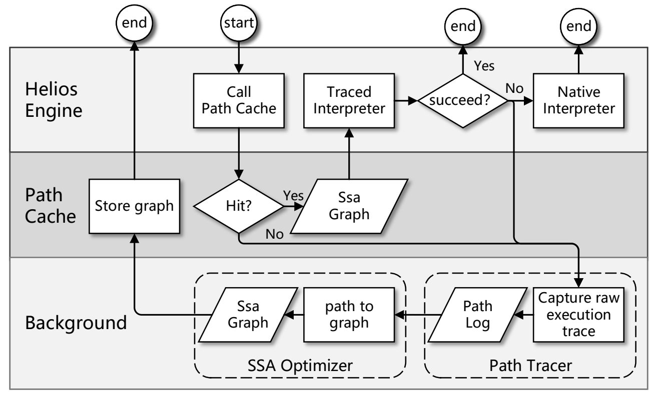

Figure 3: Overview of the Helios architecture.

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 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 allows optimization to proceed without blocking 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. This unified architecture demonstrates the *versatility* of the path-driven paradigm, enabling a single set of artifacts to serve both latency-sensitive validation and throughput-oriented historical replay.

3.2 Key Data Structures

Helios represents and indexes transaction paths through a small set of data abstractions that govern 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 inspired by Static Single Assignment (SSA) form [22]. In SSA, each value is defined exactly once, establishing a direct correspondence between values and their defining operations. This property makes data dependencies explicit, eliminating the need to track mutable stack positions and enabling classical optimizations such as constant folding and dead-code elimination. 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 [45] 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, thereby reducing overhead while preserving gas-semantic equivalence.

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
```

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 [31]. FNV-1A relies on simple multiplication and XOR operations, making it efficient for incremental updates. The resulting hash serves as a unique identifier for each execution path, enabling fast 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.

Complexity-Aware Compilation. To bound resource consumption, the optimizer enforces a configurable complexity threshold T_{max} during graph construction. If the node count of a path exceeds this limit, the optimizer aborts the transformation and marks the path as non-optimizable. This circuit-breaking mechanism prevents pathological inputs from generating excessively large graphs, ensuring that Helios focuses optimization effort on the vast majority of paths where acceleration is beneficial. Paths that exceed the threshold fall back to native interpretation, preserving correctness without degrading performance for typical workloads.

Together, the Path Tracer and SSA Optimizer form an asynchronous optimization pipeline that operates outside the critical execution path. This decoupling is central to Helios's *efficiency* objective, as it allows the system to amortize tracing and compilation costs over many subsequent invocations without penalizing the latency of individual transactions.

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

Algorithm 2: Online Mode: Path Lookup

```
Notation: S_σ 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: P: Path Cache with PML and GML layers
Output: (G, C, ℓ): Cached graph, constant table, and PathDigest; or
        ⊥ 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: P: Path Cache with PML
Output: Updated frequency statistics in S_σ
// 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)
```

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.

3.3.4 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.

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 direct-mapped virtual register file indexed by LSN. Each SsaGraph node writes its result to a dedicated slot, enabling consumers to access operands via $O(1)$ array indexing. This direct-access model eliminates the DUP/SWAP traffic that dominates stack-based interpretation, contributing to Helios's *efficiency* by removing a major source of per-instruction overhead.

While allocating a distinct register for every intermediate node may appear spatially inefficient, typical SsaGraphs remain compact due to optimization passes that aggressively eliminate redundant nodes. For the majority of execution paths, the resulting memory footprint is smaller than that of a standard EVM stack. For rare pathological cases that exceed a configurable complexity threshold, the system aborts optimization as detailed in §3.3.2, preventing memory bloat while maintaining correctness through native fallback. Empirical validation of these claims appears in §4.

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

Algorithm 4: Traced Interpreter Execution

```
Input: 𝒢 = (V, E): Optimized SSA graph
Input: C : ℕ → 𝕌_256: Constant value table
Input: Γ: Execution context (mutable)
Input: g_lim: Gas limit
Output: ⟨Γ, g_used⟩ or ⊥ (fallback)
// Initialize virtual register file and gas counter
ℛ : ℕ → 𝕌_256 ← new Array[|V|]
g_rem ← g_lim ;       // Remaining gas initialized to limit
// Execute vertices in topological order
foreach v ∈ V in topological order do
    v⃗ ← ⟨⟩ ;                                   // Operand vector
    foreach ℓ ∈ in(v) do
        if ℓ ∈ C then
            v⃗ ← v⃗ · ⟨C[ℓ]⟩ ;                  // Constant operand
        else
            v⃗ ← v⃗ · ⟨ℛ[ℓ]⟩ ;                  // Register operand
    // Control flow guard verification
    if op(v) ∈ {Jump, Jumpi} then
        pc ← ComputeTarget(op(v), v⃗, Γ)
        p̂c ← target(v) ;     // Cached target from PathLog
        if pc ≠ p̂c then
            return ⊥ ;        // Guard violation - trigger
             fallback
    // Chunked gas accounting at delimiters
    if op(v) ∈ {Gas, Return, Stop, Revert, Create, Create2}
     then
        δ_s ← chunk(v) ;   // Accumulated static gas cost
        if g_rem < δ_s then
            return ⊥ ;   // Gas anomaly - verify natively
        g_rem ← g_rem − δ_s
    // Execute opcode and update execution context
    ρ ← Exec(op(v), v⃗, Γ)
    // Deduct dynamic gas costs
    if IsDynamic(op(v)) then
        δ_d ← DynamicGas(op(v), v⃗, Γ)
        if g_rem < δ_d then
            return ⊥ ;                          // Out of gas
        g_rem ← g_rem − δ_d
    // Store result in virtual register
    if ρ ≠ ϵ then
        ℛ[ℓ(v)] ← ρ ;          // Map LSN to result value
    // Check for execution termination
    if op(v) ∈ {Return, Stop, Revert} then
        return ⟨Γ, g_lim − g_rem⟩
return ⟨Γ, g_lim − g_rem⟩
```

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. This hybrid gas model serves both *safety* and *efficiency*, preserving exact gas semantics while reducing the frequency of accounting operations on hot paths.

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 is designed to defend 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 tend to remain 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* follows 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 [5] represents high-performance Rust interpreters. We also evaluate Forerunne [20], applying its full-context tracing to both Geth and Revm. Finally, we include Revmc v0.1.0 [5], 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. Throughout all experiments, Helios maintained bit-level state consistency with the Revm baseline, empirically validating the *safety* of our hybrid execution model.

4.2 Microbenchmark Performance

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×

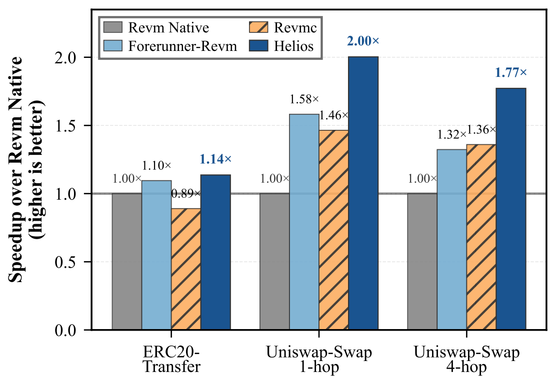

Figure 4: Execution speedup over Revm 22.0.1 baseline. Higher values indicate greater performance improvement.

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

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 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.

Table 4: Opcode Reduction and Execution Speedup

Metric	ERC20	1hop	4hop
Native opcode count	492	5,667	18,063
Opcodes Eliminated			
Constant Folding	395	4,249	13,604
Common Subexpression Elimination	10	85	257
Dead Code Elimination	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×*

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%**

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.

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. These results suggest that interpreter-level SSA specialization can achieve competitive *efficiency.*

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.

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%

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 [17] 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 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.

Figure 7 characterizes the complexity of cached SsaGraphs. The distribution is bimodal, with peaks at 201–500 nodes (30.81%) and 1K–2K nodes (26.31%). The median graph contains 494 nodes, corresponding to approximately 15.8 KB when each node stores a 256-bit value—half the 32 KB footprint of a standard 1024-slot EVM stack. This validates the register-per-node design described in §3.3.2: typical paths yield compact graphs that fit comfortably in cache. Only 0.53% of paths exceed 10,000 nodes; these trigger the complexity threshold and fall back to native interpretation.

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. The stability of storage overhead under varying thresholds also demonstrates Helios's *robustness* against path explosion, as adversarially generated cold paths are naturally excluded from the cache.

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.

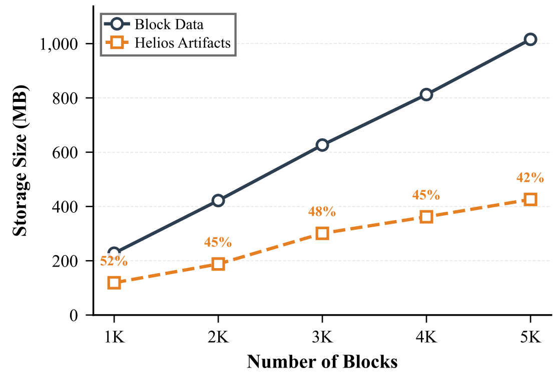

Figure 5: Storage growth for block data versus Helios optimization artifacts. Overhead decreases from 52.2% (1,000 blocks) to 41.9% (5,000 blocks) due to path convergence, where new blocks increasingly reuse existing cached artifacts.

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 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. Nevertheless, the ability to serve both throughput-oriented archival replay and latency-sensitive real-time validation from a unified artifact set highlights the *versatility* inherent in our unified architectural design. 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.

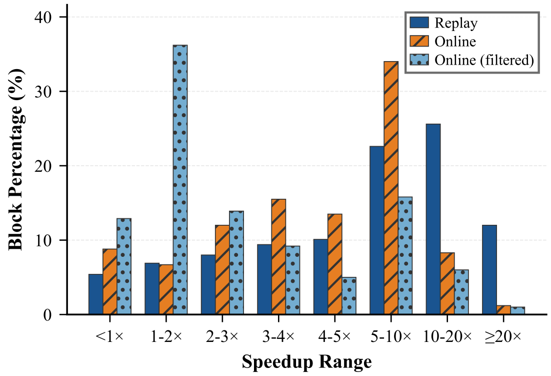

Figure 6: Block-level speedup distribution over Revm Native across three deployment modes. Replay assumes known transaction plans; Online predicts paths at runtime. Filtering ($f \geq 10$) trades median speedup for 8× storage reduction.

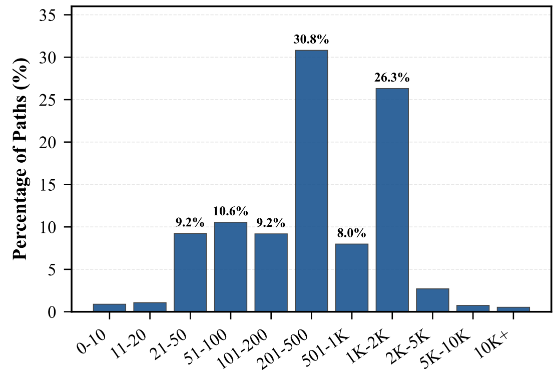

Figure 7: Distribution of SsaGraph node counts across 134,601 unique execution paths. The bimodal distribution peaks at 201–500 nodes (30.81%) and 1K–2K nodes (26.31%), with a median of 494 nodes. Only 0.53% of paths exceed 10,000 nodes.

5.1 Interpreting the Speedup

Prior work on speculative EVM execution has primarily focused on path identification as the central challenge, implicitly treating execution acceleration as a straightforward byproduct once the correct path is determined. Seer [50], for instance, concentrates on branch prediction without detailing how the predicted path translates into wall-clock savings. Forerunner [20] proposes a simplified S-EVM but does not elaborate on managing micro-architectural costs during interpretation.

Our experience suggests that achieving speedup on a highly optimized baseline such as Revm is non-trivial. Although our SSA optimization removes approximately 80% of instructions, the microbenchmark speedups range only from 1.14× to 2.00×. This gap indicates that the overhead of interpreting an intermediate representation can readily offset the gains. The memory access patterns for graph nodes and the management of virtual registers are particularly costly. Furthermore, unoptimized heavy instructions continue to dominate execution time in certain workloads.

Helios overcomes this overhead through four key design decisions that collectively yield a median speedup of 6.60× in Replay Mode. First, by tracking dependencies only for stack data, we constrain node size and minimize memory access latency. Second, frame-level granularity naturally bounds the node count and register demand per execution unit, substantially reducing memory allocation overhead compared to transaction-level tracing. Third, our SSA optimizer aggressively applies constant folding and dead code elimination, alleviating the interpreter's dispatch burden. Fourth, blocked gas metering aggregates cost deductions, further reducing the frequency of accounting operations.

We posit that surpassing the current performance ceiling may require transitioning from interpretation to JIT or AOT compilation, eliminating the virtual register abstraction entirely. The register-based design of the SsaGraph maps naturally to modern CPU architectures, facilitating this transition. Unlike traditional bytecode JITs that face "JIT bomb" risks, the acyclic and strictly bounded nature of the SsaGraph enables safe compilation without complexity explosion attacks.

5.2 Opportunities for Further Optimization

Our evaluation identifies three directions for further improvement: path selection, fallback granularity, and cache granularity.

Online Mode achieves substantial speedups in common cases, yet the gap between Top-1 prediction coverage and actual execution coverage suggests room for richer path-selection strategies. We have begun exploring a race-parallel model that caches the Top-K paths and executes them concurrently, committing the first successful result or falling back to native execution if all paths fail. Initial microbenchmarks show that coordination overhead currently limits the benefits of this approach. Lightweight synchronization primitives and speculative result caching offer promising directions to reduce this overhead.

A second opportunity lies in fallback granularity. The current transaction-scoped fallback reverts the entire transaction to native EVM upon a single frame prediction failure. A finer-grained, frame-level fallback would revert only the failed frame while continuing to use cached paths for subsequent frames, thereby increasing effective coverage. This design requires safeguards against adversarial inputs that deliberately induce frequent engine switching. Potential defenses include per-transaction fallback limits and chained predictions, where one optimized frame's output directly triggers speculative execution of the next.

A third direction involves reducing cache granularity from call frames to basic blocks. Basic-block granularity offers superior composability in principle, allowing the engine to stitch cached blocks into execution paths never observed in their entirety. However, this approach introduces a trade-off between composability and overhead. As discussed in §5.1, achieving speedup on an optimized engine is non-trivial. The cost of managing checkpoints and context switches at every basic-block boundary may exceed the savings from the optimized blocks themselves. Moreover, reconstructing consistent execution state from fine-grained checkpoints requires rigorous validation to preserve semantic correctness. Future research should investigate whether basic-block composability outweighs the administrative cost of frequent checkpointing, or whether a hybrid strategy that dynamically coalesces basic blocks into larger superblocks can balance this trade-off.

5.3 System Scalability

Beyond execution logic, system scalability presents further optimization opportunities. While our initial exploration of instruction-level parallelism (ILP) [42] showed limited improvement 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 may 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

Research on EVM acceleration spans smart contract optimization, path-driven speculative execution, and concurrent execution. Helios targets modern high-performance interpreters such as Revm, where auxiliary overhead often dominates execution time.

6.1 Smart Contract Optimization

Program Analysis. Static analysis tools such as Slither [29] and Rattle [7] generate intermediate representations for vulnerability detection and decompilation. While Rattle also lifts EVM bytecode into SSA form, it targets offline analysis without producing executable artifacts. Helios employs SSA as a dynamic executable format for immediate use by a specialized interpreter.

Superoptimization. Superoptimization frameworks [15, 16, 39] search for gas-minimal instruction sequences via SMT solvers [24]. Encoding EVM semantics including memory expansion and dynamic gas costs causes exponential blowup, restricting these approaches to small basic blocks without memory side-effects.

JIT and AOT Compilation. JIT compilers such as Revmc, Monad, and EVM-JIT [4, 9, 11, 13] translate bytecode to native code to bypass the interpreter loop. For microsecond-scale EVM transactions, code generation overhead often outweighs execution speedup. Helios instead transforms paths into an abstract graph executed by a Traced Interpreter, reducing dispatch count while retaining host-level optimizations.

6.2 Path-driven Speculative Execution

Comparison with Existing Systems. Forerunner [20] generates Accelerated Programs using full-context tracing that captures stack frames, memory snapshots, and contract state. Seer [50] employs branch prediction with state snapshots to manage dependencies. Helios addresses the artifact overhead bottleneck with lightweight asynchronous tracing that records only stack operations, achieving 9.4–16.4× compression over full-context approaches.

Granularity and Gas Semantics. Prior systems cache at transaction granularity, limiting reuse to identical sequences. Helios introduces frame-level caching to exploit path locality of individual contract calls, while preserving gas equivalence by restricting optimization to static-cost instructions.

6.3 Concurrent and Parallel Execution

Operation-Level Concurrency. ParallelEVM [38] tracks data dependencies via an SSA Operation Log, enabling selective re-execution of conflicting operations. Unlike Helios, which targets single-thread path optimization, ParallelEVM focuses on concurrency control and relies on synchronous tracing with non-negligible overhead on high-performance interpreters.

Parallel Architectures. Other approaches include PaVM [28] for intra- and inter-contract parallelism, Block-STM [32] for optimistic concurrency control, and MTPU [40] for hardware-centric spatial-temporal scheduling.

Orthogonality. Helios is orthogonal to these frameworks. Its lightweight path execution engine optimizes sequential contract execution, a baseline that can be integrated into parallel systems to further maximize throughput.

7 CONCLUSION

This paper presented Helios, a path-driven execution engine that accelerates EVM transaction processing while preserving gas-semantic correctness. On modern interpreters such as Revm, three barriers impede meaningful acceleration. The *Compilation Limitation* arises when aggressive optimizations violate gas accounting. The *Optimization Dilemma* emerges when instrumentation costs exceed execution latency. The *Granularity Mismatch* occurs when transaction-level optimization yields ephemeral artifacts. Helios addresses these challenges through *hybrid execution* that restricts optimization to static-cost instructions, *asynchronous tracing* that decouples profiling from the critical path, and *frame-level caching* that exploits call locality while resisting cache-flooding attacks.

Helios advances the state of the art along four dimensions. It ensures *safety* by maintaining gas-semantic equivalence with canonical EVM, improves *efficiency* by removing instrumentation from the critical path, provides *robustness* by resisting adversarial path explosion, and demonstrates *versatility* by serving both validators and archive nodes from a unified artifact set.

Evaluation on Ethereum mainnet confirms the effectiveness of this design. Helios achieves 6.60× median speedup in Replay Mode and 2.05× in Online Mode with 4.9% storage overhead. Future work will explore selective JIT compilation, frame-level fallback mechanisms, and hybrid cache granularity. By decoupling optimization from the critical path, Helios offers a foundation for scalable EVM infrastructure.

REFERENCES

[1] 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.
[2] 2020. Overview | Uniswap. https://docs.uniswap.org/contracts/v2/overview
[3] 2021. *Polygon: Ethereum's Internet of Blockchains.* Whitepaper. Polygon Labs. https://polygon.technology/papers/pol-whitepaper Accessed: 2025-11-24.
[4] 2024. 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
[5] 2025. bluealloy/revm: Rust implementation of the Ethereum Virtual Machine. https://github.com/bluealloy/revm

[6] 2025. BNB Chain. https://docs.bnbchain.org/
[7] 2025. crytic/rattle. https://github.com/crytic/rattle original-date: 2018-03-06T20:48:37Z.
[8] 2025. DefiLlama - DeFi Dashboard. https://defillama.com
[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. Introduction | Monad Developer Documentation. https://monad-docs-lb3l7tky7-monad-xyz.vercel.app/
[12] 2025. ipsilon/evmone: Fast Ethereum Virtual Machine implementation. https://github.com/ipsilon/evmone
[13] 2025. paradigmxyz/revmc. https://github.com/paradigmxyz/revmc original-date: 2024-03-10T16:27:36Z.
[14] 2025. Polygon Knowledge Layer. https://docs.polygon.technology/
[15] 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
[16] 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
[17] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. 2011. The Keccak reference. http://keccak.noekeon.org/
[18] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, and Francesco Tiezzi. 2020. Analysis of Ethereum Smart Contracts and Opcodes. In *Advanced Information Networking and Applications*, Leonard Barolli, Makoto Takizawa, Fatos Xhafa, and Tomoya Enokido (Eds.). Springer International Publishing, Cham, 546–558.
[19] Vitalik Buterin. 2015. A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM. https://api.semanticscholar.org/CorpusID:19568665
[20] 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.
[21] Coinbase. 2023. Base: An Ethereum L2 Network. https://docs.base.org/. Accessed: 2025-11-24.
[22] 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
[23] Xiaohai Dai, Bo Zhang, Hai Jin, and Ling Ren. 2023. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (2023). https://api.semanticscholar.org/CorpusID:259092489
[24] 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.
[25] Quinn DuPont. 2017. Experiments in algorithmic governance : A history and ethnography of "The DAO," a failed decentralized autonomous organization. https://api.semanticscholar.org/CorpusID:169783921
[26] Ethereum Foundation. 2025. Ethereum Archive Node. https://ethereum.org/developers/docs/nodes-and-clients/archive-nodes/. Page last updated: October 21, 2025.
[27] Ethereum Foundation. 2025. Nodes and Clients. https://ethereum.org/developers/docs/nodes-and-clients/. Page last updated: October 22, 2025.
[28] 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
[29] 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
[30] 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
[31] 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.
[32] 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
[33] Go Ethereum Team. 2025. Sync Modes. https://geth.ethereum.org/docs/fundamentals/sync-modes. Last edited on February 18, 2025.
[34] 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.
[35] 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.
[36] Minghang Li, Qianhong Wu, Zhipeng Wang, Bo Qin, Bohang Wei, Hang Ruan, Shihong Xiong, and Zhenyang Ding. 2025. TockOwl: Asynchronous Consensus with Fault and Network Adaptability. In *IACR Cryptology ePrint Archive*. https://api.semanticscholar.org/CorpusID:276020896
[37] Minghang Li, Qianhong Wu, Yupeng Zhang, Zhipeng Wang, Bo Qin, Xuecheng Lin, and Willy Susilo. 2025. TockCuckoo: Two-phase BFT with Linearity and Responsiveness. *IEEE Transactions on Information Forensics and Security* (2025), 1–1. https://doi.org/10.1109/TIFS.2025.3636059
[38] 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.
[39] Julian Nagele and Maria Anna Schett. 2020. Blockchain Superoptimizer. *ArXiv* abs/2005.05912 (2020). https://api.semanticscholar.org/CorpusID:218596321
[40] 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
[41] Lemayian Joel Poncha, Hachem Bensalem, Ghyslain Gagnon, Kaiwen Zhang, and Pascal Giard. 2025. EVMx: An FPGA-Based Smart Contract Processing Unit. *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)* (2025), 1708–1713. https://api.semanticscholar.org/CorpusID:280401852
[42] 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.
[43] 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
[44] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2020. A Survey of Smart Contract Formal Specification and Verification. *ACM Computing Surveys (CSUR)* 54 (2020), 1 – 38. https://api.semanticscholar.org/CorpusID:221005956
[45] 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.
[46] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-Fungible Token (NFT): Overview, Evaluation, Opportunities and Challenges. *ArXiv* abs/2105.07447 (2021). https://api.semanticscholar.org/CorpusID:234742206
[47] Sam M. Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William John Knottenbelt. 2021. SoK: Decentralized Finance (DeFi). *Proceedings of the 4th ACM Conference on Advances in Financial Technologies* (2021). https://api.semanticscholar.org/CorpusID:231662215
[48] Daniel Davis Wood. 2014. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://api.semanticscholar.org/CorpusID:4836820
[49] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019). https://api.semanticscholar.org/CorpusID:197644531
[50] 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.
[51] Shijie Zhang, Jiang Xiao, Enping Wu, Feng Cheng, Bo Li, Wei Wang, and Hai Jin. 2024. MorphDAG: A Workload-Aware Elastic DAG-Based Blockchain. *IEEE Transactions on Knowledge and Data Engineering* 36 (2024), 5249–5264. https://api.semanticscholar.org/CorpusID:268831985
[52] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2021. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 47 (2021), 2084–2106. https://api.semanticscholar.org/CorpusID:204087135