Feasibility Study of Instruction-Level Pipelining within the Ethereum Virtual Machine Architecture

Gopal Ojha

Trustless Mandir Invention and Innovation Center, Kathmandu, Nepal

Abstract

The Ethereum Virtual Machine (EVM) is a stack-based virtual processor that executes smart contract bytecode sequentially. While this design ensures determinism and correctness, it inherently limits instruction throughput. This paper presents a feasibility study of instruction-level pipelining within the EVM interpreter architecture. By analyzing the internal execution flow of the EVM as implemented in the Go-Ethereum (geth) client, the study identifies the program counter dependency, particularly under jump instructions, as the principal control hazard preventing naïve pipelining. A two-stage pipelined execution model is proposed, separating opcode fetch and decode from execution and program counter update, with a feedback mechanism to preserve EVM semantics. The work focuses on architectural feasibility rather than performance evaluation and optimization, demonstrating that pipelining inside the EVM interpreter is conceptually possible under controlled synchronization. Limitations, design challenges, and future research directions are discussed.

Keywords:
Ethereum Virtual Machine, pipelining, EVM, interpreter, stack-based, virtual machine, blockchain execution

1. Introduction

Ethereum is a decentralized blockchain platform that enables programmable smart contracts executed by the Ethereum Virtual Machine (EVM)[1]. The EVM is a stack-based virtual processor designed to execute bytecode deterministically across distributed nodes. Each instruction is executed sequentially, ensuring correctness and consensus consistency. As Ethereum trans-

action volume increases, execution throughput becomes a critical concern. While transaction-level parallelism has been explored in various contexts, instruction-level optimization within the EVM interpreter remains comparatively underexplored. Pipelining is a classical computer architecture technique used to increase throughput by overlapping independent stages of execution within a single processor. This paper investigates whether instruction-level pipelining is feasible within the EVM interpreter architecture without altering EVM semantics. The objective is not to claim immediate performance gains, but to analyze architectural constraints, identify hazards, and propose a conceptual design that enables partial overlap of instruction execution stages.

2. Background

This paper thoroughly gives sight of Ethereum Virtual Machine design, implementation, glance of sample simulation for understanding, and provides more space for pipelining in Ethereum Virtual Machine. This document provides, Introduction of research area, current system overview, pipelining in EVM, design of two stage pipelining on interpreter execution process, discussion on issues while design for implementation, and future work to optimize the design and implement in Production Environment. This document does not touch Just-In-Time compiler (JIT), or any compilers, but straight forward looks into two phase design and its surroundings for example and generalizes the case for n stage pipeline.

3. Motivation for Feasibility Study of Pipelining in EVM Architecture

Ethereum Blockchain transactional events are independent so that any stage of pipeline is possible from fetching a transaction to creating a receipt. So, taking advantage of the technique of parallelism on a single processor, Ethereum gains scale in terms of transaction throughput. Given that EVM instructions conceptually pass through distinct stages—fetch, decode, execute, and update—there exists an opportunity to overlap these stages using pipelining. In principle, pipelining can increase instruction throughput by allowing multiple instructions to be in different stages of execution concurrently on a single processor. However, due to the dynamic nature of the program counter, particularly under jump instructions, the feasibility of such pipelining is non-trivial.

4. Problem Statement

The primary research question addressed in this work is whether instruction-level pipelining is feasible within the Ethereum Virtual Machine interpreter architecture while preserving correctness and determinism. This study focuses specifically on the interpreter execution loop, rather than transaction-level or block-level parallelism.

5. Ethereum Virtual Machine Overview

The EVM is a stack-based virtual machine that executes bytecode instructions defined by the Ethereum Yellow Paper. Each instruction operates on a stack, memory, and persistent storage, while consuming gas to bound computation. The execution context includes a program counter (PC), which determines the location of the next opcode to execute. In existing implementations such as geth, the EVM interpreter executes instructions sequentially using a single program counter. The PC increments linearly for most instructions and is modified explicitly by jump instructions (JUMP, JUMPI).

5.1. Sequential Execution Model

In the geth interpreter, execution proceeds[8,10,11] as follows: Fetch opcode using the program counter. Decode the opcode and retrieve its operation metadata. Perform memory expansion and gas checks. Execute the operation. Update the program counter. Repeat until a halting condition is reached. The relevant code structure includes:

```
op = contract.GetOp(pc)
operation := in.cfg.JumpTable[op]
res, err = operation.execute(&pc, in, callContext)
pc++
```

This execution loop is strictly sequential and single-threaded.

5.2. Current system overview

EVM is based on one program counter which is reset on every new transaction. Program counter increments by one on each instruction execution except the jump operation “JUMPI “ .Fetch and decode is shown in the following code [8][9] snippet.

```
op = contract.GetOp(pc)
operation := in.cfg.JumpTable[op]
% memory checks for resizing are carried out for further
    operation.
if operation.memorySize != nil {
memSize, overflow := operation.memorySize(stack)
if overflow {
return nil, ErrGasUintOverflow
    }
% memory is expanded in words of 32 bytes. Gas is also
    calculated in words.
memorySize, overflow = math.SafeMul(toWordSize(memSize)
    ,32);
if overflow {
return nil, ErrGasUintOverflow
    }
}
```

Finally, the operation is passed to the execution function, it returns a result of the execution. The code snippet is below,

```
    res, err = operation.execute(&pc, in, callContext)
```

This result is the result of execution and needs to update the pc as:

```
case !operation.jumps:
pc++
```

And loos run forever until hulting condition arises.

6. Related Research

EVM control flow includes instructions like JUMP and JUMPI that directly mutate the program counter based on stack values. Resolving these jump targets statically is a known challenge because jump destinations may not be determinable at compile time. Advanced tools such as EVMLiSA [2] use abstract interpretation and stack tracking to reconstruct sound control-flow graphs (CFGs) by resolving possible jump targets from stack state approximations. This CFG reconstruction is crucial for high-level analyses such

as smart contract verification. However, while CFG reconstruction tools provide a static analysis approach to understanding EVM control flow, they do not address execution-level architectural optimization, such as overlapping fetch and execute stages or handling runtime pipelining hazards. Stack machine pipelining in computer architecture has inherent difficulties due to limited instruction independence in stack operations. In stack machines, unlike register machines, it is harder to insert independent instructions between dependent ones because operations are tightly coupled through the stack interface, making it difficult to schedule around hazards without exposing microarchitectural state. This architectural characteristic explains why stack interpreters such as the EVM do not readily benefit from straightforward instruction-level pipelining in the way hardware CPUs can. There is no prevalent work that successfully implements instruction pipelining inside a stack VM interpreter without changing the underlying semantics or instruction set.

Although there is no previous work on pipelining within the EVM interpreter, there are other high-level approaches to improving EVM execution efficiency that illustrate how researchers and developers have tried to circumvent sequential interpreter limitations, such as Monad [3], which aim to make Superscalar Pipelining for EVM by pipelining and parallel execution at the transaction and execution engine level, enabling throughput that can scale to thousands of transactions per second. Monad's approach is not modifying the EVM interpreter itself, but rather building an architecture where execution and state access stages are decoupled and overlapped. These approaches highlight that the only mainstream attempts to "pipeline" EVM execution focus on transaction-level or engine-level pipelining, rather than pipelining opcodes within the interpreter loop. Our feasibility study tackles this interpreter-level pipelining question directly, which existing approaches do not.

Other research explores architectural alternatives to the EVM for execution efficiency, offering context for why interpreter pipelining is still rare:

RISC-V Integration with EVM [4]: Some research explores replacing or compiling EVM bytecode to RISC-V to leverage hardware instruction pipelines on real CPUs. While this work does not pipeline inside the EVM interpreter, it demonstrates a broader strategy where EVM semantics are mapped onto a pipelining-friendly instruction set.

zkEVM Constraint Design[5]: Research into zero-knowledge Ethereum VMs investigates how different constraint architectures impact execution and proof performance. This work focuses on formal constraint engineering, not pipelining interpreter instructions.

DTVM Stack[6]: A next-generation VM design integrates hybrid lazy JIT and intermediate representations to accelerate execution, reflecting broader efforts to optimize smart contract execution through compilation and runtime techniques — rather than interpreter pipelining.

These efforts acknowledge EVM interpreter limitations and seek performance improvements by changing representation or execution strategy, but they do not explore concurrent interpreter stage overlap or control hazard handling within the interpreter loop.

The gaps exist to improve the performance in the EVM architecture optimization. Stack VM pipelining research generally focuses on hardware CPU models and higher-level scheduling, not on interpreting a blockchain virtual machine like the EVM. Existing EVM optimization proposals either focus on static analysis (e.g., CFG reconstruction) or on alternative execution environments (e.g., RISC-V, transaction pipelining), but not on instruction pipeline feasibility inside the interpreter. No prior work explicitly identifies the EVM program counter control hazard as the bottleneck for pipelining nor proposes a feedback-based pipeline model preserving EVM semantic correctness. Because of these gaps, there is no established reference architecture for instruction-level pipelining in EVM interpreters prior to this study.

The comparision of the work with the other existing related work is shown in table 2.

This feasibility study directly addresses the interpreter-level pipelining question that prior work has not. This model analyzes the EVM interpreter's semantics and identifies the program counter feedback loop as the principal control hazard, a topic not studied in CFG reconstruction or execution engine pipelining work. Instead of static analysis or alternative ISA compilation, this work proposes a two-stage interpreter pipeline with a feedback mechanism to preserve correctness under jump instructions. Whereas projects like Monad pipeline at overall execution stages, this work focuses on opcode stage overlap inside the interpreter, a level of granularity not addressed in existing work. This work fills the specific gap of instruction-level pipelining feasibil-

Table 1: Comparison of Pipelining Approaches in EVM and Related Work

Approach	**Key Features**
EVMLiSA / CFG tools	- Optimization Level: Static Analysis - Modifies EVM Semantics: No - Addresses PC Control Hazard: Yes (Static Reconstruction) - Interpreter-Level Pipelining: No
Monad	- Optimization Level: Transaction / Execution Engine - Modifies EVM Semantics: No - Addresses PC Control Hazard: Avoided via Execution Reordering - Interpreter-Level Pipelining: No
RISC-V EVM	- Optimization Level: Instruction Set / ISA - Modifies EVM Semantics: Yes - Addresses PC Control Hazard: Handled by Hardware Pipeline - Interpreter-Level Pipelining: No
zkEVM architectures	- Optimization Level: Constraint / Proof System - Modifies EVM Semantics: Yes (Formal Execution Model) - Addresses PC Control Hazard: Explicitly Encoded - Interpreter-Level Pipelining: No
This Work	- Optimization Level: Instruction Execution - Modifies EVM Semantics: No - Addresses PC Control Hazard: Yes (Runtime Feedback) - Interpreter-Level Pipelining: Yes

ity in the EVM interpreter, grounded in semantics rather than emulation or architectural redesign.

7. Feasibility overview

If we consider that the execution cycle takes only one cycle for each, and a total of four cpu cycles : Fetch opcode, decode opcode, execute opcode, and update program counter . In this case, all latencies are due to the implemented code, where there are some checks and updates of gas and memory. The current system implementation of EVM execution is prior to the release of the block. If there exist hundreds of thousands of transactions to be executed, then the time required to execute the total transactions takes significant delay. To optimize the throughput of the EVM Pipelining is one approach. There is much more research in pipelining technique. Researchers have implemented a register-based and stack-based processor. Stack-based processors in virtual machines are implemented in interpreters. Interpreter optimization and stack-based processor optimization are related in a VM environment.

7.1. Pipelining in EVM

From the execution model[7] we classified the following six stages of different operations, from transaction fetch to storing the operand in storage.

Table 2: Classifying the phases and stages of transaction in the Ethereum Virtual Machine

Phase 1 : Fetch and Decode Transaction	Phase 2 : Instruction cycle (State machine cycle)
Transaction fetching from the queue (FT)	Fetch opcode (FO)
Decode transaction (DT)	Decode Opcode(DO)
	Execute Opcode (EO)
	Store operand (SO)

An asynchronously generated transaction from a DApp is received by the node of the network. Let there be N transactions in the queue, and time ti is the start of EVM execution (assumed all transactions received before this time i). Let T be the time to execute one transaction sequentially on average. For n number of transactions, EVM takes T*N transactions execution time in total. This clearly shows that execution time is linearly dependent upon the number of transactions.

The transaction is called to the EVM, and then it first separates the bytecode and arguments. Bytecode is ready to execute as instruction operation code in the state machine. On the other hand, the argument is pushed into the stack. These arguments are operand that needs in the execution phase in the state machine cycle.

The first task of the state machine is to fetch the instruction. After fetching the instruction, the instruction contains a number of opcodes. Each opcode is executed at a time sequentially. Deduct the fees for gas for the opcode and passed for the execution; there are some opcodes that do not need any fees to execute, although all opcodes are treated here as gas consumers. In execution of opcode, opcode decides whether to read operand from stack, from the EVM state, or memory, and store in memory, storage, or stack to back. After execution is completed, again the state machine starts another cycle with another opcode and repeats continuously. Cycle stops by following the possible cases,

- Gas deficiency
- Fee deficiency
- Exception on operation

Here, we assumed, for all opcodes, there is a sufficient amount of fee to pay for gas available, and Gas is sufficiently provided to execute the opcode. This means the state machine cycle stops at the end of the opcode in the instruction.

7.1.1. Halt Avoidance

As EVM execution is a Quasi-Turing Complete where instruction should halt after a certain execution. This is needed by design to avoid the execution of the instruction forever and restrict the execution of the next instruction set. Gas is the Halting parameter and depends on the opcode. Each opcode consumes some amount of gas. Gas value is fixed and different for various opcodes.

7.1.2. EVM Phases

From the above execution model, we decipited following six sequential stages of different operations, from transaction fetch to storing the operand in storage, as shown in Figure 1.

Now we show how two-stage pipelining works as a feasibility study of the EVM execution.

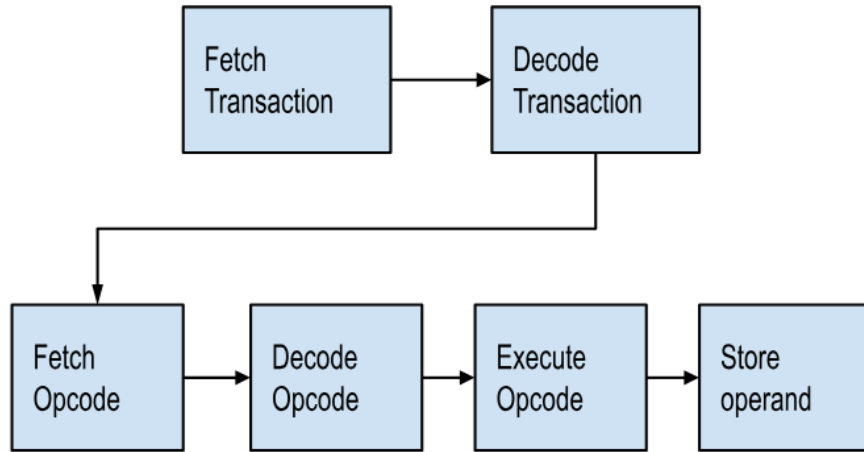

Figure 1: Depicts the six stages of the sequential execution of an opcode.

7.2. Two-Stage Pipeline Design

Pipelining requires the set of processes to work independently to accomplish an independent task concurrently by a single processor. Above defined sequential process is broken down into two stages to make two-stage pipelining. The proposed model decomposes the instruction execution loop into two stages:

Stage 1: Opcode fetch and decode
Stage 2: Instruction execution and program counter update

In this design, Stage 1 produces decoded operations that are consumed by Stage 2. Ideally, Stage 1 can proceed independently while Stage 2 executes the previous instruction.

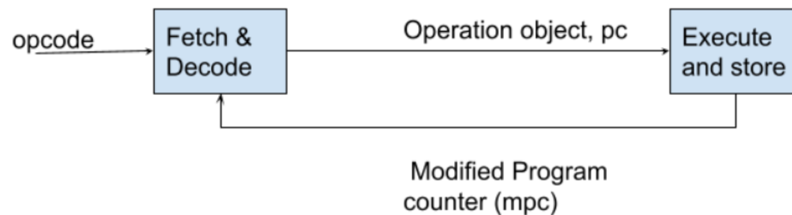

Figure 2: Two-stage pipeline Modeling.

In Figure 2, Opcode fetches and decodes to produce an output operation

object, which is consumed by the execute function. Eventually, update the program counter and provide a modified pc to fetch a new opcode.

This approach avoids resource leakage, such as waiting to update the program counter by the previous phase. EVM instructions use jump instructions to provide the opcode location, which is known after the execution of the opcode. To run the EVM instructions, it requires an updated program counter from the previous phase to fetch the opcode.

7.2.1. No Feedback Mechanism

No resource leakage if both stages work independently with respect to pc. However, current instruction sets or interpreter designs do not have such an operation to predefine the jump location. Once the jump location is synchronized using any methodology in such independent stages, pipelining benefits in the EVM-based architecture. The whole design issues of the pipelined structure depend on the synchronization of the program counter. Synchronization with the clock cycle is not included in this research. Fetch always knows the next pc by mpc, but when a jump arises, no jumped pc is tracked by fetch until the executed result updates the pc as shown in Figure 3. And this method is infeasible until modification on existing architecture.

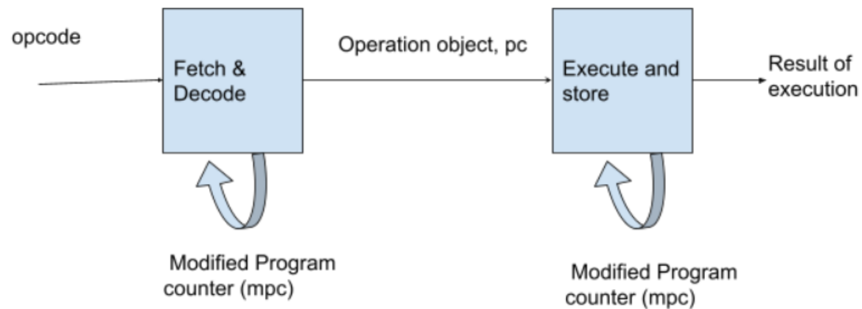

Figure 3: Sample of two-phase pipelining without feedback to the previous stage.

7.3. Feedback-Based PC Synchronization

PC value is passed to the fetch stage to update the next opcode to fetch. This creates a resource leakage at the time of waiting for a new pc. Leakage can be minimized with the prediction for triggers waiting for a pc update

event. The prediction is simply whether the opcode fetched is a jump instruction or not. If a jump, then the next opcode is located at the address defined by the jump. Hence jump should be tracked at the fetching of the opcode. If Opcode is a jump, then wait for the next pc update from the next stage, which is the execution phase. At the same time, execution completes and provides feedback as a modified counter, as shown in the Figure 4.

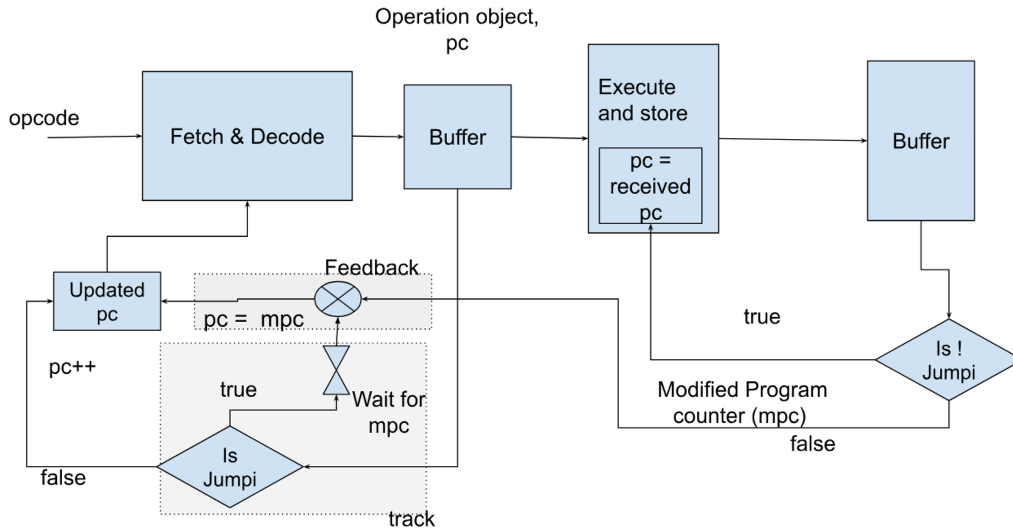

Figure 4: Update program counter using track and feedback methodology, Where Track is part of phase 1 and feedback is part of phase 2.

7.4. Sample Implementation without perfect Synchronization between stages

Here defined sample code is for the Sample Pipelining using receive only blocking channel between stages.

7.4.1. Only forward buffering

Here, feedback is not maintained. To maintain the perfect feedback synchronization, we have to Synchronize the stages with the actual frequency of opcode executions.

```
%operation channel pass operation to next phase.
operation_channel := make(chan *operation)
%pc channel pass pc to next stage
pc_channel := make(chan uint64)
```

```
5  %below closing channel after each transaction execution
      Complete or return the main thread.
6  defer close(operation_channel)
7  defer close(pc_channel)
8  %Waitgroup to synchronize the two phases so that not to
      exit the execution until both stage completes its work
       for each transaction
9
10 %wg is used for synchronization of stage 1 and stage 2,
      so that wait by stage 2 until it receives the decoded
      operation from stage 1.
11 var wg sync.WaitGroup
12 wg.Add(1)
13 %fetch_finalwg is used to not terminate stage 1 until all
       opcodes or the end of execution appear.
14 var fetch_finalwg sync.WaitGroup
15 fetch_finalwg.Add(1)
16 %fetch_finalwg is used to not terminate stage 2 until all
       opcodes or the end of execution appear.
17 var exec_finalwg sync.WaitGroup
18 exec_finalwg.Add(1)
```

For reference, the detailed code is in Appendix A.

7.4.2. Modified PC update

Modified PC should be sent as feedback in this methodology to accurately locate the next actual opcode to fetch. This is required only in the case of a jump operation. Without focusing on the number of different jump opcodes in this research, modeling is done with feedback and track, and implemented as just updating the new pc to fetch by the execute phase (phase 2 in this methodology), irrespective of any number of intermediate stages.

```
1 %mpc channel sends mpc when jumps arise
2 mpc_channel := make (chan uint64)
3 %at the beginning of stage 1.
4 %receive modified pc after jump instruction
5 fpc := <-mpc_channel
6 %at the end of phase 2.
7 case operation.jumps:
8 mpc_channel <- epc
```

The above shown channel provides the updated program counter in phase 2 (epc) is now updated in phase 1 as a fpc using channel $mpc_channel$. Here requires synchronization to pass the signal not to fetch the instruction until mpc received for jumping instruction.

8. Relating design and implementation

Process duration can be of any length for each stage; an equal time difference is not the only choice. Reducing the time of execution cycle by subdividing into stages, in such a way that execution of each transaction is increased by reducing the time of execution of the transaction from fetching the transaction to the successful result of execution, with sufficient information to maintain valid transaction. This leads to a transaction pipeline. Instead of the pipeline of transactions inside the EVM, only the pipeline of bytecode execution is focused on in this research. After all the EVM environment variables are set for the execution of bytecode, the pipeline of instructions begins from fetch to the received execution result after halt. This research takes a reference from Geth, so implementation is preferred in GoLang; the actual result from the implementation of the design given in Figure 4, which is not simulated in this research. Golang provides goroutines. Goroutines are independent stages in the pipelines. Communication between each stage is with the help of a channel. To avoid resource leakage, asynchronous send and receive is preferred, with Synchronization between the communication channels and between linked stages. Goroutines return results and errors, which may need to propagate to the main thread or pass to any communicating threads. For example: decoded operation is passed to the next phase with the help of a channel.

9. Design and issues to be solved

How safe is it to update a counter on the wait and update process? Is the security association within the feedback loop necessary? Single threads of the execution cycle are divided into multiple threads and further merged into a single one. How is it safe? The resource consumption within the execution will decrease as compared to sequential processes? Synchronization between phases requires some resources to send and receive signals and values, here single core cpu always has to provide a process to loop, new variables, synchronization signals, buffering into some channels or a buffer, synchronization

between buffed values on both sender and receiver, and checking additional jump instructions on main loop with storing forever in whole many cycles adds additional cpu cycle than sequential execution? When the number of stages increases, then the resource for such activities increases, so for the same allocated resource for sequential and pipelined, the pipelined architecture is more resource-intensive. In what case is it more feasible in such stack based architecture like EVM?

10. Discussion

This feasibility study demonstrates that instruction-level pipelining within the EVM interpreter is conceptually possible if program counter dependencies are explicitly managed. The primary architectural constraint is the control hazard introduced by jump instructions. Unlike register-based architectures, the stack-based nature of the EVM increases sensitivity to execution order and synchronization correctness. As the number of pipeline stages increases, synchronization overhead grows, potentially diminishing returns.

A fundamental challenge arises from the fact that jump instructions modify the program counter dynamically. In such cases, the next instruction address is not known until the execution stage completes. Two cases are considered:

Case 1: No Feedback Mechanism
If Stage 1 proceeds without receiving updated PC values from Stage 2, incorrect instruction fetches occur whenever a jump instruction is executed. This makes naïve pipelining infeasible.

Case 2: Feedback-Based PC Synchronization
In this approach, Stage 2 sends the updated program counter back to Stage 1 when a jump instruction is executed. Stage 1 temporarily stalls fetching until the updated PC is received. This feedback mechanism preserves correctness while enabling partial pipelining.

If we observe the synchronization and resource considerations, Pipelining introduces synchronization overhead between stages. In a single-core environment, this includes:

- Communication latency between pipeline stages
- Additional control logic for detecting jump instructions

- Buffering and synchronization of program counter updates

While pipelining reduces idle time between stages, it also increases resource usage due to synchronization primitives. Therefore, feasibility depends on whether the benefits of overlap outweigh synchronization costs.

For the sample implementation approach, this study references the Go-Ethereum (geth) interpreter and proposes a conceptual implementation using Go's goroutines and channels to model pipeline stages.

For the stage communication, channels are used to pass:

- Decoded operations
- Program counter values
- Feedback PC updates in case of jumps

Execution flow is maintained as, Stage 1 fetches and decodes opcodes, Stage 2 executes operations and updates the PC, and for jump instructions, Stage 2 sends the updated PC back to Stage 1.

This implementation is illustrative and not intended as a production-ready system. Quantitative benchmarking is outside the scope of this work, as the objective is to identify architectural constraints and correctness-preserving mechanisms.

11. Conclusion

There are various research studies conducted on pipelining on bytecode-based VM interpreters with several modifications in other existing interpreters, like in JVM, which provides the successful showcase the possibility of being pipelined, the bytecode-based interpreter that works on the stack machine operation. In this research It is shown the way to pipeline inside the interpreter specially designed for EVM. It suggests after the complete measure of each instruction cycle with cpu cycle and possible latencies mitigations to reduce the instruction cycle time, pipelining in the Ethereum Virtual Machine (EVM) is feasible. This work intentionally does not claim performance improvements, provide quantitative benchmarks, modify gas accounting rules, address Just-In-Time (JIT) compilation. The focus is strictly on architectural feasibility and design constraints.

This paper presents a feasibility study of instruction-level pipelining in the Ethereum Virtual Machine architecture. By analyzing the interpreter

execution loop and identifying program counter dependencies as the primary control hazard, a feedback-based two-stage pipeline design is proposed. While synchronization costs remain a concern, the study demonstrates that pipelining within the EVM interpreter is architecturally feasible under controlled conditions. This work provides a foundation for future optimization research in EVM execution engines.

12. Future Work

Instead of a go-channel approach, which is shared memory by communication, the possible buffering and accessing methodology might be a fast, less resource-consuming design, and asynchronous activities can be an efficient, deep research is required here. Simulating the actual design after implementation in different stages to make it efficient is required. Once the optimum methodology is acquired, the methodology gains importance in virtual stack-based interpreters, because stack-based interpreters are making a great impact in the Distributed and Decentralized networks. Future research directions include measuring instruction-level execution latency, evaluating synchronization overhead quantitatively, exploring alternative buffering mechanisms, extending the design to multi-stage pipelines, assessing feasibility in production-grade EVM clients

Appendix A. Sample Code for Two Stage Pipelining

Here is a sample code for showing how two-stage pipelining can be applied in EVM architecture.

```
 %operation channeel pass operation to next phase.
operation_channel := make(chan *operation)
%pc channel pass pc to next stage
pc_channel := make(chan uint64)
defer close(operation_channel)
defer close(pc_channel)
var wg sync.WaitGroup
wg.Add(1)
var fetch_finalwg sync.WaitGroup
fetch_finalwg.Add(1)
var exec_finalwg sync.WaitGroup
exec_finalwg.Add(1)
```

```
13      go func() {
14              % enough stack items available to perform the
                  operation.
15              op = contract.GetOp(fpc)
16              operation := in.cfg.JumpTable[op]
17              wg.Done()
18              operation_channel <- operation
19              pc_channel <- fpc
20              switch {
21                      case !operation.jumps:
22                          fpc++
23              case operation.jumps:
24             % wait until stage 2 send updated pc to fetch
                 stage to continue new fetch from jump
                 location
25              }
26            }
27          fetch_finalwg.Done()
28      }()
29      go func() {
30              wg.Wait()
31              wg.Add(1)
32              operation := <-operation_channel
33 a            epc := <-pc_channel
34             % execute the operation
35              res, err = operation.execute(&epc, in,
                  callContext)
36              switch {
37        % many cases
38              case !operation.jumps:
39                  epc++
40              case operation.jumps:
41        % sends updated pc to fetch stage to continue new
             fetch from jump location
42              }
43          }
44          exec_finalwg.Done()
45      }()
46      fetch_finalwg.Wait()
47      exec_finalwg.Wait()
```

```
48
49
50 }
```

References

[1] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform," 2013. Available: https://ethereum.org/en/whitepaper/, accessed Feb. 9, 2026.

[2] V. Arceri, S. M. Merenda, L. Negrini, L. Olivieri, and E. Zaffanella, "EVMLiSA: Sound Static Control-Flow Graph Construction for EVM Bytecode," *Blockchain: Research and Applications*, vol. 100384, 2025. Available: `https://www.sciencedirect.com/science/article/pii/S2096720925001113`, doi: `https://doi.org/10.1016/j.bcra.2025.100384`.

[3] Reflexivity Research, "Monad Overview," Available: `https://www.reflexivityresearch.com/all-reports/monad-overview`, accessed Feb. 9, 2026.

[4] RISC-V EVM: A Novel Approach to Blockchain Virtual Machine Architecture, Available: `https://github.com/developeruche/riscv-evm-experiment`, accessed Feb. 9, 2026.

[5] Y. Hassanzadeh-Nazarabadi and S. Taheri-Boshrooyeh, "Constraint-Level Design of zkEVMs: Architectures, Trade-offs, and Evolution," arXiv:2510.05376v1, 2025. Available: `https://doi.org/10.48550/arXiv.2510.05376`, accessed Feb. 9, 2026.

[6] W. Zhou et al., "DTVM: Revolutionizing Smart Contract Execution with Determinism and Compatibility," arXiv:2504.16552, 2025. Available: `https://doi.org/10.48550/arXiv.2504.16552`, accessed Feb. 9, 2026.

[7] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," Ethereum Project Yellow Paper, 2014. Available: `https://ethereum.github.io/yellowpaper/paper.pdf`, accessed Feb. 9, 2026.

[8] Go-Ethereum Project. Available: https://github.com/ethereum/go-ethereum, accessed Feb. 9, 2026.

[9] Ethereum Developer Documentation. Available: https://ethereum.org/en/developers/docs/evm/, accessed Feb. 9, 2026.