Bridging Worlds: A Performance and Feasibility Analysis of RISC-V Integration with Ethereum Virtual Machine

Full Draft Report

Full Draft Report code: https://github.com/developeruche/riscv-evm-experiment/tree/main/crates/research-draft

Developeruche
https://x.com/developeruche

Abstract

This research explores the technical feasibility and performance implications of integrating the RISC-V instruction set architecture with the Ethereum Virtual Machine (EVM). As blockchain platforms seek improved efficiency and accessibility, this work examines whether RISC-V's open architecture can provide a viable alternative execution environment for smart contracts. Through experimental implementation and benchmarking, we evaluate both the technical challenges of this integration and its potential benefits, including multi-language smart contract development, execution efficiency, and architectural compatibility. My findings indicate that while RISC-V integration presents promising opportunities for blockchain development diversity, significant challenges remain in reconciling RISC-V's register limitations with complex EVM operations and preserving the security guarantees of traditional EVM environments. This work contributes to the ongoing discourse on blockchain virtual machine evolution and provides empirical insights into the future architectural directions for Ethereum and similar platforms.

1. Introduction

Ethereum's Virtual Machine (EVM) remains the cornerstone of its smart contract functionality, providing a sandboxed execution environment that has enabled a diverse ecosystem of decentralized applications. However, as blockchain technology matures, questions emerge about the long-term viability and limitations of the current EVM architecture. This research investigates an alternative approach by exploring RISC-V, an open instruction set architecture, as a potential foundation for executing smart contracts. The convergence of blockchain technology and RISC-V presents intriguing possibilities. RISC-V's open nature, simplicity, and growing ecosystem make it an attractive candidate

for reimagining blockchain execution environments. By enabling smart contracts to be written in languages that target RISC-V, this integration could potentially broaden the accessibility of blockchain development while potentially addressing performance and security considerations inherent in the current EVM design.

This research addresses several fundamental questions:

1. Can RISC-V effectively implement the functionality required for EVM-compatible smart contract execution?
2. What are the performance implications of this architectural shift?
3. How might this integration affect the developer experience and ecosystem?
4. What fundamental design challenges must be overcome for practical implementation?

2. Background and Related Work

2.1 The Ethereum Virtual Machine (EVM)

The EVM is a quasi-Turing complete, 256-bit stack machine [Cite EVM Spec]. Its key components include a stack, volatile memory, and persistent storage. Execution proceeds by interpreting bytecode instructions (opcodes) that manipulate these components. Gas mechanics regulate computational cost. While successful, criticisms often focus on its performance compared to native execution, the complexity of its 256-bit word size on standard 64-bit hardware, and the learning curve associated with Solidity and the EVM's specific execution model [Cite EVM critiques].

2.2 The RISC-V ISA

RISC-V is an open-source ISA based on reduced instruction set computing (RISC) principles [Cite RISC-V Specs]. Its base integer ISA (e.g., RV32I, RV64I) is small and can be extended with standard extensions (e.g., 'M' for integer multiplication/division, 'A' for atomics, 'C' for compressed instructions). This modularity allows tailoring processors for specific needs. A significant advantage is the mature and growing compiler support (GCC, LLVM), enabling compilation from various languages.

2.3 Alternative Blockchain Execution Environments

Several projects have explored alternatives to the EVM. EOS and Polkadot utilize WebAssembly (WASM) as their execution engine, aiming to leverage its performance potential and language flexibility [Cite Polkadot WASM, EOS WASM]. Solana employs a custom architecture optimized for parallel execution, using LLVM to compile contracts written in languages like Rust and C [Cite Solana VM]. These efforts highlight the ongoing search for more performant and developer-friendly blockchain execution layers.

2.4 Bridging EVM and Other Architectures

Some research has focused on JIT (Just-In-Time) compilation for the EVM or transpiling Solidity to other languages or bytecode formats [Cite EVM JIT efforts]. Our work differs by exploring the direct implementation of EVM semantics using a fundamentally different ISA (RISC-V) as the target.

3. Methodology: Designing and Implementing a RISC-V EVM

To achieve these goals, the applied methodology would be broken into a two-phase approach:

Phase One: Custom RISC-V EVM Architecture

1. Design of the core components of this new RISC-V EVM VM, this would also invovle blockchain related operations fashioned as `opcodes` rust implementation
2. Implementation of a RISC-V IM32 assembler compatible with blockchain requirements. This would be task to assemble RISCV assemble written smart-contracts to RISCV machine code code.
3. Optimization of the implementation for performance, security specs constraints.
4. Benchmarking and preliminary analysis

Phase Two: Integration with Existing Ethereum Runtime

1. Adaptation of custom RISC-V EVM to the REVM (Rust Ethereum Virtual Machine) API.
2. Implementation of blockchain-specific operations (Excecution of blocks and transactions) within RISC-V constraints.
3. Comparative performance testing between custom implementation and standard EVM
4. Analysis of register utilization and optimization opportunities

3.1 Phase One: Custom RISC-V EVM Implementation

1. **Architecture Design:** We chose the RV32IM subset of the RISC-V ISA as the target, providing basic integer operations and multiplication/division, deemed sufficient for initial EVM opcode mapping. The design focused on creating a RISC-V interpreter capable of managing state analogous to the EVM's (stack, memory, simulated storage access). During this phase blockchain related operations fashioned as ecalls it ignored. Here is the code implementing this phase. *It is important to disclose this code was implemented before this research idea was concived, making this implementation not connected to the blockchain in any way which is exactly the point.*

2. **EVM Opcode Mapping:** Each relevant EVM opcode was mapped to a sequence of RV32IM instructions. Simple arithmetic opcodes (`ADD` , `SUB`) translated relatively directly, while stack manipulations (`PUSH` , `POP` , `DUP` , `SWAP`) required careful management of RISC-V registers or memory to simulate the EVM stack. More complex opcodes presented significant challenges:
    - `SSTORE` : Requires interaction with the storage mechanism and involves multiple parameters, straining register availability.
    - `LOG*` : Opcodes like `LOG2` , `LOG3` , `LOG4` consume numerous inputs (memory offset/length, topics), leading to high register pressure, sometimes exceeding available registers in a simple mapping.
    - `CALL` variants: Involve managing call frames, arguments, and return values, demanding complex sequences of RISC-V instructions.

3. **RISC-V Interpreter/Assembler:** A basic interpreter for the chosen RV32IM instruction subset was implemented in Rust. This interpreter managed the RISC-V register file and simulated memory access. A rudimentary assembler was developed to convert mapped EVM logic (represented as RISC-V instruction sequences) into executable format for the interpreter. code

4. **Handling Blockchain Context:** Direct interaction with blockchain state (storage, account balances, block information) is essential. We designed an abstraction layer using simulated **"environment calls"** (ecalls in RISC-V terminology). This approach mirrors the concept outlined in the `counter_riscvim32_smart_contract_asm` example, where specific `ecall` instructions trigger host functions to retrieve context information (e.g., `env::block_number()` , `storage::get()`). This isolates blockchain-specific logic from the core RISC-V execution of contract logic but necessitates developers using these specific library calls. see

3.2 Phase Two: Adaptation to REVM API

To facilitate comparison and leverage a mature EVM framework, we adapted our custom RISC-V execution core to fit within the REVM architecture. REVM provides well-defined traits and structures for EVM components like the `Interpreter` , `Host` , and `Database` .

1. **Integration:** I implemented REVM's `Interpreter` trait, replacing its core instruction loop with calls to our RISC-V interpreter for executing the mapped EVM bytecode (now represented as RISC-V code).
2. **State Management:** REVM's `Database` and `Host` interfaces were used to handle state lookups (storage, balances, code) required by the execution, replacing the

simpler simulation used in Phase One. This allowed interaction with more realistic EVM state representations.

3. **Goal:** The primary goal of this phase was to enable more direct (though still preliminary) comparisons by running the same contract logic through both a standard REVM interpreter and our RISC-V-based interpreter operating within the same REVM framework.
   This phase is to be completed on the final research result, result from phase two would be found there.

4. Experiementation and test on draft implementation

For the purpose of this research a draft implementation of this VM have been implemented and a `riscv_assembler` also code can be found here;

1. Draft RISVIM32-EVM: https://github.com/developeruche/riscv-evm-experiment/tree/main/crates/research-draft/riscv_evm/src
2. RISCVIM32 assembler: https://github.com/developeruche/riscv-assembler

To test the compactiablity of the VM, a simple counter smart contract write as RISCV assemble, was deployed and excecuted on this VM.
Here is what this smart contract looks like;

```
# SPDX-License-Identifier: MIT
# Simple Counter Contract in RISC-V RV32IM Assembly

# --- Ecall Definitions (Using the same as provided) ---
.equ ECALL_KECCAK256, 0x20    # [offset, size] -> hash
.equ ECALL_ADDRESS, 0x30      # Address of the current executing contract |-> ad
.equ ECALL_BALANCE, 0x31      # Native balance of the current caller [address] -
.equ ECALL_ORIGIN, 0x32       # Address of the transaction origin |-> address
.equ ECALL_CALLER, 0x33       # Address of the current calling address |-> addre
.equ ECALL_CALLVALUE, 0x34    # Deposit value for this Tx |-> value
.equ ECALL_CALLDATALOAD, 0x35 # Load a Word(256bits) from the calldata [i] -> da
.equ ECALL_CALLDATASIZE, 0x36 # Returns the size of the calldata |-> usize
.equ ECALL_CALLDATACOPY, 0x37 # Copy calldata from input to memory [destOffset,
.equ ECALL_CODESIZE, 0x38     # Returns the size of the code |-> usize
.equ ECALL_CODECOPY, 0x39     # Copy code from input to memory [destOffset, offs
.equ ECALL_GASPRICE, 0x3A     # Gas price now
.equ ECALL_EXTCODESIZE, 0x3B  # Get the size of an External account's code [addr
.equ ECALL_EXTCODECOPY, 0x3C  # Get the code of an External account [address, de
.equ ECALL_RETURNDATASIZE, 0x3D # Get size of output data from the previous call
.equ ECALL_RETURNDATACOPY, 0x3E # Copy output data from the previous call [destO
.equ ECALL_EXTCODEHASH, 0x3F  # Get hash of an account's code [address] -> hash
.equ ECALL_BLOCKHASH, 0x40    # Get hash of recent block [blockNumber] -> hash
.equ ECALL_COINBASE, 0x41     # Get the block's beneficiary address |-> address
.equ ECALL_TIMESTAMP, 0x42    # Get the block's timestamp |-> timestamp
.equ ECALL_NUMBER, 0x43       # Get the block's number |-> blockNumber
.equ ECALL_PREVRANDAO, 0x44   # Get the block's difficulty |-> difficulty
.equ ECALL_GASLIMIT, 0x45     # Get the block's gas limit |-> gasLimit
.equ ECALL_CHAINID, 0x46      # Get the chain ID |-> chainId
```

```
.equ ECALL_SELFBALANCE, 0x47  # Get balance of currently executing account |-> ba
.equ ECALL_BASEFEE, 0x48      # Get the base fee |-> baseFee
.equ ECALL_BLOBHASH, 0x49     # Get versioned hashes [index] -> blobVersionedHasl
.equ ECALL_BLOBBASEFEE, 0x4A  # Returns the blob base-fee of the current block |-
.equ ECALL_GAS, 0x5A          # Amount of available gas
.equ ECALL_LOG0, 0xA0         # Append log record with no topics [offset, size]
.equ ECALL_LOG1, 0xA1         # Append log record with one topic [offset, size, t
.equ ECALL_LOG2, 0xA2         # Append log record with two topics [offset, size,
.equ ECALL_LOG3, 0xA3         # Append log record with three topics [offset, size
.equ ECALL_LOG4, 0xA4         # Append log record with four topics [offset, size,
.equ ECALL_CREATE, 0xF0       # Create a new account with code [value, offset, si
.equ ECALL_CALL, 0xF1         # Call into an account [gas, address, value, argsOf
.equ ECALL_CALLCODE, 0xF2     # Call with alternative code [gas, address, value,
.equ ECALL_RETURN, 0xF3       # Halt execution returning output data [offset, siz
.equ ECALL_DELEGATECALL, 0xF4 # Call with alternative code, persisting sender and
.equ ECALL_CREATE2, 0xF5      # Create account with predictable address [value, o
.equ ECALL_STATICCALL, 0xFA   # Static call into an account [gas, address, argsOf
.equ ECALL_REVERT, 0xFD       # Halt execution reverting state changes [offset, s
.equ ECALL_SLOAD, 0x54        # Loads a word (32-bytes) from storage
.equ ECALL_SSTORE, 0x55       # Stores a word (32-bytes) to storage


# --- Storage Slot Definitions ---
.equ SLOT_COUNTER_1, 0
.equ SLOT_COUNTER_2, 0
.equ SLOT_COUNTER_3, 0
.equ SLOT_COUNTER_4, 0
.equ SLOT_COUNTER_5, 0
.equ SLOT_COUNTER_6, 0
.equ SLOT_COUNTER_7, 0
.equ SLOT_COUNTER_8, 0


.text           # Entry point for deployment (initcode)

# ============================================================
# INITCODE SECTION
# Runs only once during deployment.
# Sets up initial state and returns the runtime code.
# ============================================================
_start:
    # --- Initialize Counter Value to 0 ---
    addi x1, zero, SLOT_COUNTER_1
    addi x2, zero, SLOT_COUNTER_2
    addi x3, zero, SLOT_COUNTER_3
    addi x4, zero, SLOT_COUNTER_4
    addi x5, zero, SLOT_COUNTER_5
    addi x6, zero, SLOT_COUNTER_6
    addi x7, zero, SLOT_COUNTER_7
    addi x8, zero, SLOT_COUNTER_8

    addi x9, zero, 0
    addi x10, zero, 0
    addi x11, zero, 0
    addi x12, zero, 0
    addi x13, zero, 0
    addi x14, zero, 0
    addi x15, zero, 0
```

```
    addi x16, zero, 1

    # Store initial counter value to storage
    addi x31, zero, ECALL_SSTORE
    ecall

    # --- Return Runtime Code ---
    # Calculate the size and offset of the runtime code section
    addi x5, zero, runtime_code_start # Get address of runtime code start
    addi x6, zero, runtime_code_end

    sub x3, x6, x5            # x3 = length of runtime code

    # Return the copied code
    add x1, zero, x5               # mem_offset = x5
    add x2, zero, x3               # length of runtime code
    addi x31, zero, ECALL_RETURN
    ecall

    # End of initcode. Should not be reached after ECALL_RETURN.


# ================================================================
# RUNTIME CODE SECTION
# This code is stored on the blockchain after deployment.
# It handles subsequent calls to the contract.
# ================================================================
runtime_code_start:
    # --- Runtime Entry Point & Function Dispatcher ---
    # Read function selector (first 4 bytes of calldata)
    addi x1, zero, 0               # offset = 0
    addi x31, zero, ECALL_CALLDATALOAD  # setup for the ECALL
    ecall                     # Returns first 32 bytes: Note in this case x2 woul


    # Function dispatcher - compare with known selectors
    addi x3, zero, 0x00000037   # Selector for increment()
    beq x2, x3, _increment

    addi x3, zero, 0x00000020   # Selector for getValue()
    beq x2, x3, _getValue

    addi x3, zero, 0x00000055   # Selector for setValue(uint256)
    beq x2, x3, _setValue

    # Fallback: If no function matches, revert
    jal x0, _revert_default

# --- Function Implementations ---

_increment:
    # Increment the counter value by 1
    # 1. Load current counter value
    # 2. Add 1
    # 3. Store new counter value
    # 4. Return success (true)

    # Load current counter value
```

```
    addi x1, zero, SLOT_COUNTER_1
    addi x2, zero, SLOT_COUNTER_2
    addi x3, zero, SLOT_COUNTER_3
    addi x4, zero, SLOT_COUNTER_4
    addi x5, zero, SLOT_COUNTER_5
    addi x6, zero, SLOT_COUNTER_6
    addi x7, zero, SLOT_COUNTER_7
    addi x8, zero, SLOT_COUNTER_8

    addi x31, zero, ECALL_SLOAD  # setup for the ECALL
    ecall                    # Returns counter value in [x9 - x16]


    # Increment counter (need to handle potential overflow from any of the limbs)
    addi x16, x16, 1            # counter_last_limb += 1, next is to check for ov
    beq x16, zero, _inc_overflow      # If counter_last_limb wrapped to 0, increm
    jal x0, _inc_store

_inc_overflow:
    addi x15, x15, 1            # counter_last_limb-1 += 1
    bne x15, zero, _inc_store      # If counter_last_limb-1 does not wrapped to 0

    # Increment counter_last_limb - 2
    addi x14, x14, 1            # counter_last_limb-2 += 1
    bne x14, zero, _inc_store      # If counter_last_limb-2 does not wrapped to 0

    # Increment counter_last_limb - 3
    addi x13, x13, 1            # counter_last_limb-3 += 1
    bne x13, zero, _inc_store      # If counter_last_limb-3 does not wrapped to 0

    # Increment counter_last_limb - 4
    addi x12, x12, 1            # counter_last_limb-4 += 1
    bne x12, zero, _inc_store      # If counter_last_limb-4 does not wrapped to 0

    # Increment counter_last_limb - 5
    addi x11, x11, 1            # counter_last_limb-5 += 1
    bne x11, zero, _inc_store      # If counter_last_limb-5 does not wrapped to 0

    # Increment counter_last_limb - 6
    addi x10, x10, 1            # counter_last_limb-6 += 1
    bne x10, zero, _inc_store      # If counter_last_limb-6 does not wrapped to 0

    # Increment counter_last_limb - 7
    addi x9, x9, 1            # counter_last_limb-7 += 1
    bne x9, zero, _inc_store      # If counter_last_limb-7 does not wrapped to 0,




    # some missing code see the link below;




    # Store initial counter value to storage
    addi x31, zero, ECALL_SSTORE
    ecall
```

```
    # Return success (true = 1)
    jal x0, _return_true

# --- Helper Routines ---

_return_true:
    # Prepare return value 'true' (uint256(1))
    addi x2, zero, -48           # Allocate stack space
    addi x31, zero, ECALL_SSTORE
    addi x3, zero, 0
    addi x4, zero, 1

    sw x3, 0(x2)              # Store counter_limb_1 at sp
    sw x3, 4(x2)              # Store counter_limb_2 at sp+4
    sw x3, 8(x2)              # Store counter_limb_3 at sp+8
    sw x3, 12(x2)             # Store counter_limb_4 at sp+12
    sw x3, 16(x2)             # Store counter_limb_5 at sp+16
    sw x3, 20(x2)             # Store counter_limb_6 at sp+20
    sw x3, 24(x2)             # Store counter_limb_7 at sp+24
    sw x4, 28(x2)             # Store counter_limb_8 at sp+28

    # Return true u256(1)
    add x1, zero, x2                # mem_offset = 0
    addi x2, zero, 32                # length = 8 bytes (uint256)
    addi x31, zero, ECALL_RETURN
    ecall

    # Clean up stack (should not be reached after ECALL_RETURN)
    addi x2, x2, 8
    jal x0, _return_true

_revert_default:
    addi x1, zero, 0
    addi x2, zero, 0
    addi x31, zero, ECALL_REVERT
    ecall

runtime_code_end:              # Mark the end of the runtime code section
```

Complete Counter RISCV smart contract: code

In order to deploy this contract and make a call here is the API design for this Draft implemenation for obtaining this goal;

```
use revm::{
    Context as RevmEthContext, DatabaseCommit, MainContext,
    context::{ContextTr, JournalTr},
    database::CacheDB,
    primitives::{Address, U256},
};
use riscv_assembler::assembler::assemble;
use riscv_evm::{
    context::Context,
    ecall_manager::process_ecall,
    riscv_evm_core::{MemoryChuckSize, e_constants::*, interfaces::MemoryInterfac
```

```
    utils::{bytes_to_u32, u32_vec_to_address, u32_vec_to_bytes},
    vm::Vm,
};
mod contract;

fn main() {
    let from_addr = Address::from([
        0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66,
        0x00, 0xAA, 0xBB, 0xCC, 0xDD,
    ]);

    let mut vm = Vm::new();
    let eth_context = RevmEthContext::mainnet().with_db(CacheDB::default());
    let mut context = Context::new(eth_context);

    context.current_caller = from_addr;

    // ==================================
    // deploying the counter contract
    // ==================================
    let init_code = assemble(contract::ASSEMBLE_CODE).unwrap().code;
    let init_code = u32_vec_to_bytes(&init_code, init_code.len() * 4);
    let init_offset = 900;

    // Write init code to memory
    for (i, &byte) in init_code.iter().enumerate() {
        vm.memory
            .write_mem(init_offset + i as u32, MemoryChuckSize::BYTE, byte as u3
    }

    // Set up Create ECALL
    vm.registers.write_reg(ECALL_CODE_REG, 0xF0); // Create
    vm.registers.write_reg(CREATE_INPUT_REGISTER_1, init_offset);
    vm.registers
        .write_reg(CREATE_INPUT_REGISTER_2, init_code.len() as u32);

    // Set value (0.0..01ETH)
    let value = 10_000_000_000_000_000_000u64;
    let value_bytes: [u8; 32] = U256::from(value).to_be_bytes();
    for i in 0..8 {
        vm.registers.write_reg(
            CREATE_INPUT_REGISTER_3 + i as u32,
            bytes_to_u32(&value_bytes[i * 4..(i + 1) * 4]),
        );
    }

    // Load account of the creator so it can transfer value
    context
        .eth_context
        .journal()
        .load_account(context.address)
        .unwrap();

    // Going ahead to excecute this ecall
    let result = process_ecall(&mut vm, &mut context).unwrap();

    // Commit all states changes to database
    for i in result {
```

```
            context.eth_context.db().commit(i.state);
        }

        // Check that output registers were written
        let addr1 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_1);
        let addr2 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_2);
        let addr3 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_3);
        let addr4 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_4);
        let addr5 = vm.registers.read_reg(CREATE_OUTPUT_REGISTER_5);

        // Reconstruct address to check it's valid
        let address_bytes = u32_vec_to_address(&[addr1, addr2, addr3, addr4, addr5]);
        println!("Address: {:?}", Address::from(address_bytes));

        context
            .eth_context
            .journal()
            .load_account(Address::from(address_bytes))
            .unwrap();
        let new_contract = context
            .eth_context
            .journal()
            .load_account_code(Address::from(address_bytes))
            .unwrap()
            .clone()
            .info
            .code
            .unwrap();
        println!("This is the runtime code: {:?}", new_contract); //NOTE!!!: this ru
    }
```

The aim of the draft implemenation is to see that there is little to no change on the main Ethereum protocol in acheiving this complete change of the the excecution sandbox (EVM -> RISCV-EVM), storing the RISCV bytecode in the same location (Account code section) where the EVM bytecode is stored, replacing `stack` related operations with `register` operations and so on.

5. Draft Result

After the implementation and test of the draft RISCV-EVM here are some implementation based results;

1. Simple arithmetic, logic, and stack operations generally mapped efficiently to RISC-V instructions, showing plausible execution times within the interpreter framework.
2. The already provided `Context` using in EVM implementations like revm plugs in fine with draft implementation.
3. 44 enviromental calls including (Keccak256, Address, Call, Create, Create2 and so on). all ecalls
4. Some ethereum opcodes almost consumes all the registers of the RISV EVM (examples SSTORE, LOG2, CALL), sometimes, it actually comsumes all, needing more registers

for it operation (LOG3, LOG4) making opcode like that not possible with registers as it temporal storage.

computational results would available in the final draft

6. Discussion

6.1 Interpreting the Results

The preliminary findings suggest that while using RISC-V for EVM execution is feasible, achieving performance parity or superiority over optimized EVM implementations faces hurdles. The overhead observed, particularly for complex opcodes suffering from register pressure, indicates that a naive mapping of EVM semantics to RISC-V may not yield significant performance gains without substantial optimization efforts (e.g., JIT compilation, ISA extensions tailored for EVM operations) which were beyond the scope of this initial experiment. The lack of gas metering further complicates direct performance comparisons based on economic cost.

6.2 Developer Experience Revisited

One motivation for exploring alternative VMs is to attract developers familiar with mainstream languages. RISC-V, with its LLVM backend, theoretically enables this. However, this experiment underscores a critical point: writing the *logic* of a smart contract in a language like Rust is only part of the challenge. Developers must still understand and interact with the blockchain's unique context – storage layout, transaction semantics, account models, gas economics, and security considerations. Abstraction libraries (like the conceptual `riscv-evm-utils`) are necessary but represent a new layer of specific APIs developers must learn. Therefore, while the language barrier might be lowered, the *blockchain domain knowledge* barrier remains substantial. It is not "all roses"; significant learning is still required.

6.3 Architectural Implications and Future Directions

This experiment highlights a fundamental architectural crossroads:

- **Option A: High-Fidelity EVM Emulation on RISC-V:** Sticking closely to EVM semantics ensures compatibility but, as observed, can lead to architectural friction and potential performance overhead on RISC-V. Overcoming this might require complex JIT compilers or custom RISC-V extensions specifically designed to accelerate EVM operations, potentially compromising the generality of RISC-V.
- **Option B: Native RISC-V Blockchain Design:** Abandoning strict EVM compatibility allows for a design that leverages RISC-V's strengths natively. This could involve rethinking blockchain storage, addressing schemes, account models, and signature

verification to align better with a register-based architecture. While potentially much more performant and cleaner architecturally, this represents a radical departure, sacrificing compatibility and requiring a massive ecosystem transition.

This leads to profound questions about Ethereum's evolution: Are we tethered to the EVM's design philosophy indefinitely? What will Ethereum's execution layer look like in 10-20 years? Is the burden of such fundamental innovation feasible for core developers, or will it inevitably be pushed to Layer 2 solutions, potentially increasing fragmentation? These are difficult, politically charged questions, but essential for the long-term health and scalability of the ecosystem. The desire for alternative VMs stems from the real limitations of the EVM, but the path forward is far from clear.

6.4 Limitations Revisited

The conclusions drawn here are heavily influenced by the preliminary nature of the experiment. The lack of gas metering, optimizations, and the experimental quality of the code mean that performance observations are indicative rather than definitive. A fully optimized, production-ready RISC-V EVM might exhibit different characteristics.

7. Conclusion and Future Work

This paper presented an initial experimental exploration of implementing an EVM-compatible execution environment on the RISC-V ISA. Through a two-phase approach involving a custom interpreter and integration with REVM, I investigated the feasibility and challenges.

My main conclusion is multifaceted: While RISC-V offers a pathway to potentially leverage broader language toolchains for smart contract development, it is not a straightforward solution for replacing or enhancing the EVM. I observed significant architectural friction when mapping complex EVM opcodes to RISC-V, notably register pressure, suggesting potential performance overheads in a direct emulation approach. Furthermore, the complexity of blockchain state interaction remains a significant hurdle for developers, irrespective of the contract language used. This work suggests that the path towards leveraging alternative ISAs like RISC-V for Ethereum-like blockchains forces a difficult choice between maintaining EVM compatibility with potential inherent inefficiencies, or pursuing a radical, incompatible redesign for a potentially more performant 'native' architecture.

Future Work:
Significant work is required to mature this exploration:

1. **Implement Gas Metering:** Accurately implement EVM gas calculation for RISC-V instruction sequences.

2. **Implement EVM Optimizations:** Incorporate features like EIP-2929 (storage warming).
3. **Comprehensive Benchmarking:** Conduct rigorous benchmarks against optimized EVM implementations (e.g., geth, REVM) using standard test suites.
4. **Optimization:** Explore JIT compilation techniques or optimized interpretation strategies for the RISC-V execution core.
5. **Bug Fixing and Stabilization:** Develop a robust, well-tested implementation.
6. **Explore Custom Extensions:** Investigate the potential benefits and drawbacks of designing custom RISC-V ISA extensions tailored for accelerating EVM operations.
7. **Theoretical Analysis:** Further analyze the trade-offs between EVM emulation and a native RISC-V blockchain design.

This research contributes early insights into the complex interplay between ISA design and blockchain execution environments, highlighting both the opportunities and the profound challenges in evolving platforms like Ethereum.

8. References

- Ethereum Yellow Paper: Formal EVM Specification
- MoonPay: What is the Ethereum Virtual Machine (EVM)?
- Wilcke: Optimising the Ethereum Virtual Machine
- RISC-V International: Official Website
- RISC-V Instruction Set Manual, Volume I: User-Level ISA
- RISC-V Instruction Set Manual, Volume II: Privileged Architecture
- REVM: Rust Implementation of the Ethereum Virtual Machine
- Polkadot Wiki: WebAssembly (Wasm) Documentation
- EOSIO Developer Portal: EOS VM Documentation
- Solana Developer Portal: Comprehensive Resources
- evmjit: The Ethereum EVM JIT Library

- Ethereum Foundation Blog: Go Ethereum's JIT-EVM