Ejit-EVM — A low latency JIT for the Ethereum Virtual Machine

Andy Thomason
andy@atomicincrement.com
www.atomicincrement.com

Abstract: The Ethereum virtual machine has become the de-facto execution engine for smart contracts. Significant players — such as Uniswap, Curve and AAVE have built significant businesses by running transactions on Ethereum Mainnet or on the many EVM-based Layer 2 chains.

However, the current implementations of the EVM have very low efficiency, typically requiring 1000 to 10,000 machine cycles per EVM instructions executed and experiments show us that we can reduce this to 1 or fewer cycles per instruction.

While many other JIT enviroments exist, such as LLVM, they have very high compile times which makes it very difficult to use an EVM based on LLVM in an environment where there are millions of deployyed contracts — each ERC20 has an indiviudal contract and so must be compiled. LLVM takes millions of machine cycles to compile an EVM contract and saves only a few cycles in execution which makes it impractical for use in real applications except in the case where the number of contracts is severely limited.

Ejit — on the other hand s— is a very low level Just in time compiler that takes only a few thousand cycles to compile the same contracts into machine code and can execute them immediately. It can be used with the EVM (Etehereum), eBPF (Solana) or Move (Aptos, SUI) to accelerate execution and reduce costs of hosting by several orders of magnitude.

The result should be a significant cost reduction in blockchain hosting and lower latency for transactions.

1 Introduction

Let us look at the start of the UniswapV2 contract, the most deployed contract on mainnet and hence the most executed code on mainnet.

```
6080 PUSH1 80
6040 PUSH1 40
52   MSTORE
34   CALLVALUE
```

```
80   DUP1
etc.
```

A naive interpreter, such as Revm, loops over the bytes in the contract and calls one of 256 functions to execute each opcode, such as PUSH1. The cost of this is hundreds of machine cycles per opcode with some, like MSTORE costing much more than say DUP1.

An advanced intereter would translate this code into the native machine code, say aacrch64, the least expensive CPU architecture available on many cloud servers.

Instructions like PUSH1 and DUP1 are effectively zero cost as they just modify the immediate values used by subsequent instructions.

The MSTORE, for example, can just copy a constant from the code to memory using vector instructions — two cycles on ArmV9 SVE architectures — but we must also check for a memory expansion and charge extra gas accordingly. A compare against the current memory size and a conditional call serve to do this. MSTORE-MLOAD pairs may also be eliminated through optimisation.

All instructions incur a cost of incrementing a GAS register, but this is very small and can be aggregated into a single /emphadd instruction.

The most expensive instruction is SLOAD which will usually incur an expensive call to a BTreeMap cache and in the case of a miss to a very expensive KV-store /emphget operation. The cache fetch can be alleviated by generating the fetch code in the JIT — using CRC32 instructions for a hashmap, for example, but a low-latency KV-store is still a work in progress for our team.

2 The Ejit VM

The Ejit VM is a simplified machine model based on a fusion of aarch64 and x86_64 programing models. We have two register classes, integer and vector with correpsonding instrucions. Ejit will mostly translate these 1:1 but in the case of x86_64 it is sometimes necessary to emit multiple instructions per ejit instruction.

The number of registers is variable and needs to be managed by the layer above. We do not attempt to do register allocation, relying on the translation layer to use the appropriate registers.

Instructions are generated as a Rust enum as in the following example.

```
let mut prog = Executable::from_ir(&[
    Movi(COUNT, 10000),
    Movi(TOT, 0),
    Movi(INC, 1),
Label(LOOP),
    Add(TOT, TOT, COUNT),
    Sub(COUNT, COUNT, INC),
    Cmpi(COUNT, 0),
    B(Cond::Ne, LOOP),
```

```
        Mov(RES[0], TOT),
        Ret,
    ])
    .unwrap();
```

3 Parallel execution

To facilitate parallel execution, the EVM should be implemented as a future so that operations that block, such as code fetch and compilation or state reads do not block the current tread — this is a current problem with Revm for example — this enables speculative execution of transactions in a block, rolling back transactions which are proven to have earlier dependencies — such as MEV transactions.

With this mechanism, we hope to go from the current GGas/sec (billion EVM gas cost) to TGas/sec (trillion gas per second) or approximately 300 EVM blocks per second at the current tip on a medium sized server or tens of blocks per second on a raspberry Pi-sized device. This compares to 0.1 blocks per second on the current platform on a $3000 per month server.

To achieve this, the EVM needs to be re-entrant and able to exit on blocking instructions such as SSTORE, EXTCODESIZE or CALL.

4 Revm compatibility

The current design uses Revm input data structures to enable its use in Reth and other Revm-based ethereum nodes.

This will enable third-parties to do integration with ethereum chains, for example by using Celestia as a data availability layer.

We could make the design more efficent by building a lighter weight wrapper than the current Revm data structures, but we aim for adoptability first.

5 Other chains

We can support other virtual machines to make existing blockchains more efficient. We hope to be invited to accelerate chains such as Fuel, Movement, Solana, Aptos, SUI and others. We can make proof-of-concepts for all these chains by way of introduction as we have worked with these chains in the past and are familiar with their VMs.

6 Conclusion

Ejit-evm is a small, lightweight, ethereum virtual machine that can be used to accelerate evm-like chains.

There is a strong need to make blockchain nodes cost effective and we can solve this problem by reducing the cost to host an EVM chain.

We can use the EJit core to accelerate other chains with bytecodes as well as scripting languages such as Lua, Javascript and Python. We may launch Python and R Ejit packages to facilitate AI and statistical analysis.