Numerically Intensive Deterministic Smart Contracts

Alexis Schlomer

School of Computer and Communication Sciences

Decentralized and Distributed Systems Lab

BSc Semester Project

10 June 2022

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Enis Ceyhun Alp
EPFL / DEDIS

Contents

Chapter 1

Introduction

1.1 Motivation

Seven years ago, Ethereum was launched and the world saw its first decentralized smart contract platform in operation. It held the promise of replacing what we now consider indispensable centralized intermediaries, such as banks, insurance companies and various governance structures, on a massive scale. However, there is still a significant gap between its ultimate goals and what is technically feasible today.

The reasons for this discrepancy are numerous, ranging from Ethereum's consensus model to its underlying smart contract execution platform: the Ethereum Virtual Machine (EVM). While offering crucial deterministic execution, the EVM and its associated programming language, Solidity, have specific issues that not only hinder contract execution time but also burden the development process.

1.2 Goals

The goal of this semester project is to find other deterministic alternatives to the EVM and compare them in terms of efficiency and ease of programming. The platforms of particular interest to us are the Java Virtual Machine (JVM) and WebAssembly (WASM).

Our analysis will be supported by two numerically intensive applications, which are frequently encountered and encompass a wide range of common problems. Particular interest will be given to floating points, since their deterministic support on the JVM and WASM is one of the current working points of the DEDIS lab at EPFL.[1]

Chapter 2

Background

This chapter presents the theoretical background of our project. We first present how smart contracts are currently executed deterministically, before proposing two more efficient and developer-friendly alternatives.

2.1 Current Environment

2.1.1 Smart Contracts

Smart contracts are a stream of arbitrary user-defined code. These instructions enforce the rules of an immutable software agreement. When executed, contracts change the state of the blockchain, which is itself a distributed append-only data structure.[2]

They are an effective way to establish trusted transactions between disparate parties. For this architecture to work well, consensus must be reached. All nodes must agree on the same outcome when these smart contracts finish execution. This emphasizes the importance of code **determinism**.

2.1.2 Ethereum Virtual Machine

This section succinctly explains at a high level how Ethereum, the most popular decentralized smart contract platform, executes its contract code deterministically on the EVM, along with its associated limitations.

Source of Determinism

The Ethereum protocol defines a specialized virtual machine called the EVM to run its smart contract code. The opcodes available on this virtual machine are restricted. A higher-level operating system cannot be accessed because it is not designed for generic purpose programming. An EVM opcode merely defines a transition between two states on the blockchain. As such, every

transaction can simply be viewed as a sequence of transitions that move the blockchain from its starting to destination state. Each opcode has a specific behavior and associated cost defined in detail in the Ethereum Yellow Paper. [3]

Since the EVM works at the granularity of an opcode, the smart contract code must be precompiled when deployed on the network. An interpreter is then responsible for executing this sequence of instructions. The code can be converted into machine binaries by a rudimentary Just-In-Time (JIT) compilation mechanism, at the cost of some security risks. This simplicity of design is what allows secure deterministic contract execution.[4]

Limitations

Aside from the obvious slowness of an interpreter, the EVM comes with other limitations and caveats. We decided to include the restrictions of Solidity, EVM's *de facto* standard programming language, since both are intrinsically linked and together constitute the entire development experience.

Some of the following issues will be illustrated in more depth later, during the implementation of our two numerically intensive target applications.

1. Solidity is not a generic purpose programming language. Although Turing Complete, it does not support I/O operations, does not allow you to fork your code into different processes or threads, and most importantly, does not allow you to schedule automatic executions. This is because every change in Ethereum must come from a manual transaction from an arbitrary address.

2. All operations on the EVM are executed on 256 bits, and the memory is 256 bit addressable. This creates an unusual memory management experience. In addition, this peculiar design choice creates poor interoperability with existing hardware today that typically runs on 64 bits.[1]

3. There are almost no standard libraries available, and the built-in data structures are very limited. Indeed, maps and arrays are the only defined complex data structures. Yet, their API is very limited because it depends on the underlying implementation. These two problems

create situations where the developer has to reinvent the wheel several times.

4. At a lower level, we are confronted with the lack of very useful opcodes on the EVM, especially with regard to floating points as well as cryptographic operations. Our modern hardware is equipped with specialized units that perform these operations very efficiently.[14] Therefore, these behaviors have to be emulated on the EVM, which results in a significant efficiency overhead. The reasons being that floating points do not have a fully deterministic behavior.[1]

2.2 Alternative Environments

This section explores what alternatives exist to move away from the EVM and adopt a different yet deterministic execution environment.

2.2.1 Disclaimer

At the time of writing, we are fully aware that neither the JVM nor WASM are inherently deterministic. While a subset of operations can be extracted which are, work is still in progress to allow a larger subset of operations to be deterministic, including a floating point support.

We are also fully conscious that if these were to be implemented as smart contract execution environments, it would be necessary to adapt the infrastructure to allow appropriate state transitions and to include a gas metering. In addition, the JIT compilation of these two platforms should be restricted to avoid potential vectors of attack. This will certainly impact the final efficiency and programming experience.[5]

However, these previous points are outside the scope of our work. Our analysis will be done under the assumption that these problems have been solved and what can consequently be gained.

2.2.2 Java Virtual Machine

The Java Virtual Machine (JVM) offers a well-established high-performance alternative. Its main underlying language, Java, is very popular and has

many existing libraries and implementations ready for use. Java is a generic purpose object-oriented programming language with which most developers are familiar.

2.2.3 WebAssembly

WebAssembly (WASM) is our second proposal since it is increasingly popular for web applications and has near-native code execution efficiency. It also offers the developer the advantage of choosing its preferred generic purpose programming language, such as C, C++ or Go.[5]

In this project we decided to opt for C, to enable more powerful low-level features. C is also historically one of the first languages supported on WASM. WASM bytecode can be generated from C source files using the Emscripten compiler and linker. The Node.js runtime environment can then execute the produced output.

Chapter 3

Designs & Implementations

This chapter presents the design and implementation of two **numerically intensive** applications that both represent practical and promising showcases, but that are currently unable to scale properly on the EVM.

3.1 Methodology

The next two sections are each devoted to one application. Each section will briefly introduce the application motivations and use cases, before diving into an analysis of runtime and ease of programming across the three previously introduced platforms: EVM, WASM and JVM.

3.1.1 Benchmarking

Benchmarking was done on a standard computer with the following specifications.

- **CPU**: AMD Ryzen 7 2700X Eight-Core Processor
- **Memory**: 24 GB RAM

The benchmark measures the latency of every transaction in nanoseconds. To have comparable results, the benchmark on every environment is directly embedded inside the source code of the executed contract. Our complete setup can be found online on our Github repository.[6]

Some specific challenges of various difficulties had to be overcome on every platform, so we include hereunder a succinct explanation for each environment.

EVM

This environment was by far the most difficult to evaluate, as we cannot simply invoke the EVM to run our smart contract. We must first launch a local Ethereum client like Geth and instantiate a local blockchain before deploying our contract. Only then can we issue a transaction. Fortunately, we found an online open-source benchmarking tool called bsol that simulates this entire pipeline and sped up the whole process.[7]

We instrumented Geth's source code to only benchmark the interpreter execution. By doing this, we avoided the unnecessary time spent on other pipeline stages. We decided to print the total gas cost and breakdown for all instructions to emphasize where most of the time was spent. The execution time for each sample was averaged over 10 iterations.

However, we encountered many problems in achieving this. First, we should mention that each block on Ethereum has a maximum capacity of 30 million gas.[3] As we pushed the EVM up to its limits, this artificial boundary had to be removed. While this was not problematic at first, it had a major and unexpected repercussion. Indeed, before broadcasting a transaction, the Geth client pre-executes it several times locally to estimate the ideal gas cost. This is done by a binary search whose upper bound is the maximum gas capacity per block mentioned above. Thus, by increasing the upper bound, the pipeline remained stuck in the gas estimation phase. As a solution, we suppressed this estimation in the Geth pipeline.

Despite all of this instrumentation, we still experienced some process crashes and errors when approaching transactions with gas costs in the billions that took several seconds to execute. This is most likely due to other limitations such as the maximum stack call depth of 1024.[3] We have left these problems as they are, since they can be equated to another limitation of the EVM.

JVM

We decided to use the Java Microbenchmark Harness (JMH) framework to measure the efficiency of our code. We set up one warm-up iteration of the JVM before sampling the results. We took the average execution time over 10 iterations, equally split over two forks.[8]

WASM

A customized benchmark framework has been implemented which includes one warm-up iteration. We used the wall clock time to measure elapsed time for consistency with other environments. As with Solidity and Java, we took the average execution time over 10 iterations.

3.2 Revenue Distribution

3.2.1 Design

The first application concerns a revenue distribution protocol. Revenues are to be distributed to users based on their participation in an asset pool. We illustrate this with the following non-exhaustive real-world applications.

- ◇ **Decentralized lending platforms**

  This notorious Decentralized Finance (DeFi) application can be viewed as follows: some people lend assets and represent the stakeholders in our framework, while other people borrow assets and are generating an income stream for the lenders through the interests they pay.

- ◇ **Liquidity pools for decentralized exchanges**

  Another well-known use case on DeFi is decentralized exchanges based on liquidity pools instead of order books. Users deposit both assets of an exchange pair in a pool and collect revenue in the form of a trading fee when a swap is initiated.

- ◇ **Decentralized prediction markets**

  Although more hypothetical than the previous use cases, it is nevertheless presented to show the potential scope of a generic revenue distribution protocol. Decentralized prediction markets are currently primarily focused on solving discrete case outcomes and distributing rewards on a winner-takes-all basis. However, when considering continuous outcomes, we introduce the notion of outcome proximity. This leads to more refined distribution functions, where we have to take into account the prediction of all participants and perform a weighted redistribution of bets.

In fact, revenue distribution is already occurring in several situations and can often be deployed with a scalable and efficient solution. However, these implementations are very specific and can hardly be generalized to equivalent use cases.[9]

Thus, our proposed protocol has very loose assumptions in order to be as modular as possible. In particular, we allow the following configurations.

- The revenue and stake asset may not be the same. In the case of a lending application, this means that interests can be paid in another currency than the loan.
- The user stakes and the total stake can both be dynamic, so no assumptions can be made on the constancy of a user stake ratio.

For this project, we provide two approaches to tackle this problem: one naive and one optimized.[9]

1. **Naive Version**

   The naive approach can simply be seen as an iteration over every user when revenue is being distributed. However, this incurs a complexity cost of $\mathcal{O}(users)$, since the implementation boils down to a simple loop.

2. **Optimized Version**

   The optimized approach we developed uses a lazy calculation technique combined with storing a well chosen global state and well chosen user states. This achieves a $\mathcal{O}(1)$ complexity, but requires more **complex floating-point arithmetic**.[9]

3.2.2 Implementation

This subsection reviews the **ease of programming** of the optimized and naive implementation we encountered on Solidity, before comparing it with its equivalent version on Java and C. The code of each implementation can be found online, on our Github repository.[6]

Solidity

We will focus on three major non trivial implementation particularities we had to overcome on Solidity for this application.

1. **The absence of floating points**

   The EVM has no support of floating points. Often these can be emulated by fixed point arithmetic on integers up to 256 bits wide. The developer must identify the ideal number of bits to allocate to the fractional part. In our case, we determined that allocating 60 bits to the fractional part was ideal, since around 60 bits are needed to represent the fractional part of a standardized ERC20 token. This left us with only 196 bits for the integer part. While this may seem like a lot, this only translates into a maximum representable number of 10^{59}, as opposed to the usual IEEE-754 standard maximum number of 10^{308}.

   As a result, we often experienced overflows when multiplying two large numbers. Therefore, we had to introduce a new auxiliary function which temporarily extends the multiplication to 512 bits using two 256 bit integers. A division by a large number would then bring the number back into the 256 bit range. We recall that it would have been impossible to start with the division before multiplying to avoid overflow, since this would have resulted in a significant loss of precision.

   This highlights the classic problem that can be encountered when working with fixed point numbers. By keeping a fixed space between each representable number, we not only reduce our maximum range, but we also lose the ability to represent extremely small numbers. Although this solution is feasible for small applications, it represents an additional workload for the programmer since it is very easy to make mistakes.

```
// Our aforementioned function signature (optimized version)
function mulDiv(uint256 a, uint256 b, uint256 denominator)
    private pure returns (uint256 result)
```

2. **256 bits arithmetic**

All the arithmetic on the EVM is done on 256 bits. As an example, the bool type, which should theoretically take one bit, also takes 256 bits. This turns out to be a significant problem, especially when it comes to storing in the expensive persistent memory. As a solution, Solidity offers the possibility to store numerous fields inside a C-like structure, whose size will be capped to the upper 256 bit multiple. While this technique brings significant performance gains, it moves a big burden to the programmer. Indeed, the developer must determine what the most fitting bit width of each field is.

A related issue concerns the poor interoperability between operands of different widths. Indeed, an explicit cast to uniformize the operands width must be performed at every operation to avoid any potential overflows. Additionally, similar casts must be done when composing signed and unsigned types.

```
// This structure packs together fields of different widths
// Each field encapsulates a part of a user's state
// These fields can be used to find the revenue lazily
struct UserState {
    uint144 ownStake;
    uint144 lastTotalStake;
    uint144 lastIncrementPerRevenue;
    uint112 ownAccumulatedTotal;
    uint256 lastIndex;
}
// Here is a code snippet that illustrates the aforementioned
// bad type interoperability
function changeShare(int256 _amount) public {
    // [...]
    userState.ownStake =
        uint144(uint256(int256(uint256(userState.ownStake)) +
        _amount));
}
```

3. **Limited data structures**

   The only complex data types available in Solidity are lists and maps. Having a limited API, we had to overcome the fact that maps do not provide a *contain* method, nor a *size* attribute. Also, there is no way to iterate over all keys and values in the map. While this is due to implementation constraints on Patricia Merkle Trees, it definitely complicates the developer's task who must add counterintuitive redundant data structures to mitigate this problem.[10]

```
// Code snippet available in the naive implementation
uint public totalHolders;
mapping(uint => address) public users;
mapping(address => UserState) public usersStateMap;
```

Java

All of the fore mentioned problems with Solidity do not appear in the Java implementation. This is mostly due to the presence of a floating point support. In addition, Java has no problem composing different types together.

```
// As a comparison, this is the equivalent valid code in Java
public void changeShare(String dest, double change) {
    // [...]
    userData.ownStake += change;
}
```

We can note the presence of an additional parameter *dest* in the function to compensate for the absence of a *msg.sender* in the global API. If a true Java-based smart contract development framework were to emerge, we believe this value would be available as an attribute of a super class. See **section 4.2** for more details.

Java also offers rich and efficient standard libraries to the developer. This turned out to be of particular relevance when it came to choosing a fitting map implementation which would suit our requirements. In our case, we

identified that a *HashMap* would best suit our needs, since it offers an efficient random access for user stake changes as well as an efficient iterator to loop over the values of the map when distributing.

```
// Thus, our code gets reduced as follows
// A string can be viewed as an alias to an address
private final Map<String, UserData> userDataMap;
```

C

In our situation, C offered the same advantages as Java over Solidity. We could use floating points and did not have to worry about type interoperability. Although C offers better memory control than Java, this proved to be more of a burden than an advantage in these implementations.

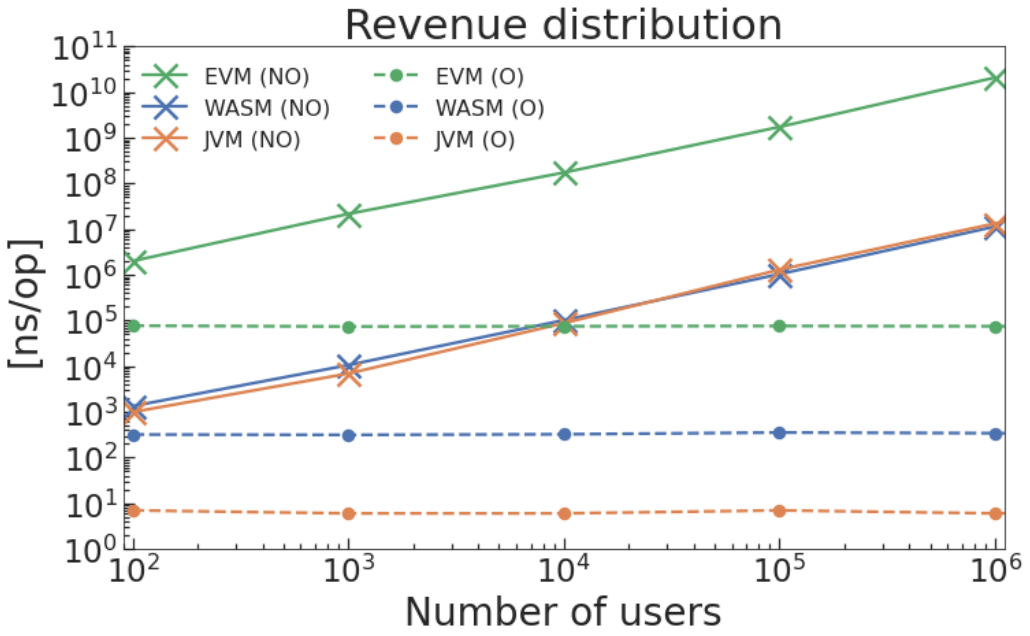
Like Solidity, C does not have an extensive standard library that the developer can use. This had a considerable impact on our work, since we had to find an equivalent and efficient hashmap implementation in C. Therefore, we decided to use an existing, open-source hashmap implementation found online.[11] Although this implementation is optimized for random access, its iterator over values was found to strongly underperform Java's by a factor of **5x**. After careful analysis, we realized that this was due to poor cache locality when iterating over all the values in the map, since each key had to be hashed before accessing the underlying value, creating a sporadic random access sequence. We solved this problem by adding a redundant linked-list data structure optimized for iteration over the set of values.

Unlike Java, we had to compile and link the code manually using a tool called Emscripten to generate WASM bytecode. Fortunately, this was straightforward to do, since our application does not employ any external libraries that would require prior translation into WASM bytecode.

3.2.3 Evaluation

Results

Following the configuration of **section 3.1.1**, we get the following results for the optimized version (labeled O) and the naive version (labeled NO) on our three different execution environments.



First, we can observe that the optimized versions across all platforms take time $\mathcal{O}(1)$ as opposed to the naive version which takes time $\mathcal{O}(users)$.

As expected, both the JVM and WASM heavily outperform the EVM. The slowdown ranges from a factor of circa **2,000x** for the naive approach, **250x** for the optimized approach assuming WASM and **10,000x** assuming the JVM.

We include the following tables that breakdowns the total gas cost of a distribution transaction involving 1,000 users on both versions. For the naive approach, we excluded opcodes with a total gas cost of less than 5,000.

OPCODE	OCCURRENCES	TOTAL GAS
ADD	3,000	9,000
DIV	1,000	5,000
DUP1	2,007	6,021
DUP2	7,002	21,006
DUP4	3,000	9,000
JUMP	1,006	8,048
JUMPI	2,009	20,090
MSTORE	4,001	12,012
MUL	1,000	5,000
PUSH1	10,008	30,024
PUSH2	3,015	9,045
SHA3	2,000	84,000
SHL	1,000	3,000
SLOAD	5,001	4,000,800
SSTORE	1,000	800,000
SWAP1	6,000	18,000
SWAP2	3,000	9,000
SWAP3	2,000	6,000
TOTAL	**64,083**	**5,072,117**

Table 3.1: Gas for $1,000$ users (NO)

OPCODE	OCCURRENCES	TOTAL GAS
ADD	1	3
DIV	0	0
DUP1	6	18
DUP2	1	3
JUMP	6	48
JUMPI	6	60
MSTORE	1	12
MUL	1	5
PUSH1	11	33
PUSH2	12	36
SHA3	0	0
SHL	1	3
SHR	1	3
SLOAD	2	1,600
SSTORE	1	20,000
SWAP1	3	9
SWAP2	0	0
SWAP3	2	6
TOTAL	**80**	**21,889**

Table 3.2: Gas for $1,000$ users (O)

As we can observe, more than **95%** of the total gas cost originates from persistent memory loads (SLOAD) and stores (SSTORE). These accesses act as the bottleneck of the execution.

JVM and WASM performance are almost equivalent for the naive version, the slight variations can be explained by the different rehashing thresholds of the two hashmap implementations.

The factor **50x** between the optimized JVM version and WASM can be explained by different benchmarking implementations. Indeed, our custom WASM benchmarking framework has a higher constant time overhead than JMH. This difference gets noticeable when dealing with small values. Thus, the execution time overhead of the EVM compared to WASM is likely to be greater than **250x** for the optimized version.

3.3 Neural Networks

3.3.1 Design

The second application tackles an important field in machine learning: Deep Neural Networks (DNNs). Although completely unrealizable as of today due to the infrastructure's limitations, implementing DNNs on decentralized blockchains such as Ethereum does have promising applications. We illustrate this with the following list of non-exhaustive use cases.

- ◇ **Finance**

  In Decentralized Finance (DeFi), we can use DNNs to improve the liquidity and profits of an Automated Market Maker (AMM) using an optimal IA-driven strategy.[12].

- ◇ **Authentication**

  Publicly verifiable deterministic IA models like DNNs can be used as an authentication method to link public addresses with real people. We can imagine a signature or fingerprint recognition algorithm to establish a proof of personhood.

- ◇ **Reproducibility**

  Reproducing the same results on different computers of the same machine learning algorithm with identical input data is not guaranteed. Yet, reproducibility is crucial in science. Introducing determinism would therefore solve this problem.[13]

We made the decision to design a limited deep neural network to stay in the scope of this project. In particular, we implemented a classification contract that uses a multilayer perceptron neural network. The rectified linear unit (ReLu) is used as intermediate activation function. Stochastic gradient descent is applied to train the model.

When deploying the contract, the user specifies the dimensions of the hidden layers along with the training and test data. The designed architecture supports an arbitrary number of features and classes. A special function call has been integrated to normalize the input data before training.

To best fit the scope of this project, data samples are immediately converted internally to their floating point representation. In the case of Solidity, we used the ABDK floating point library which emulates deterministic quad precision floating point numbers.[15] This library was not used in the previous application, as we could use the more efficient fixed point arithmetic. However, this application requires a very extensive numerical range since it involves extensive gradient calculations and matrix multiplications when backpropagating samples.

3.3.2 Implementation

This subsection reviews the **ease of programming** of the neural network implementation on Solidity, before comparing it with its equivalent version on Java and C. Once again, the code of each implementation can be found online, on our public Github repository.[6]

Our implementation follows Michael Nielsen's approach in his online book *Neural Networks and Deep Learning.*[16]

Solidity

Implementing a functional neural network on Solidity was not an easy task, given the complete absence of any existing matrix library. As a consequence, we not only had to code a matrix library from scratch, but we also had to mock Java's pseudo random number generator. The reason being that smart contracts on Solidity are as hard to test as they are hard to code. Thus, we started by implementing the neural network on Java and using that as a control version.

In order to train efficiently, we initialized the model weights to pseudorandom values. While this can be done easily with the Java *Random* class, there is no such support on Solidity. Thus, we implemented Java's linear congruential generator to get reproducible results across all platforms. In both cases, the seed is hard-coded to zero, since generating sources of randomness is orthogonal to the notion of determinism.

We opted for a functional programming paradigm to make the code clearer and more understandable. Thus, every function in our matrix library pro-

duces a new object in memory, without ever modifying the input data. Solidity references these functions with the special *pure* keyword.

```
// The signature of a pure function in our matrix library
// W is thus left unmodified
function dAffineDx(M.Matrix memory W) internal pure returns
    (M.Matrix memory)
```

Although this choice might sound like a good programming practice, this is a bad idea on Solidity. Indeed, the EVM specifies no garbage collector like the JVM, and cannot free temporary heap memory. Thus, every allocation gets appended at the end of a continuous memory strip, only to get freed when the transaction completes.[3]

We nevertheless took this route, since the other alternatives would have made the code totally incomprehensible.

Java

Although time-consuming, the Java implementation went without any noteworthy troubles. We used EJML as linear algebra library, which proved to be convenient and efficient. Nevertheless, the library still lacks some functions that had to be coded in a dedicated utility file.[17]

In addition, EJML follows a functional programming paradigm which made the implementation even more straightforward. Unlike the EVM, the JVM disposes of a garbage collector that efficiently frees up unreferenced space in memory.

Finally, we have integrated the MNIST dataset in our implementation for testing purposes. We coded a custom interface to load a percentile of the data in the correct format.[19]

C

For this implementation, we used GSL as linear algebra library. Unlike EJML, this library only offers a limited API to the developer. As a consequence, even more basic functions had to be coded separately.[18]

Furthermore, GSL distinguishes vectors from matrices. This made the implementation more cumbersome as vectors had to be transformed to matrices and vice-versa. Additionally, GSL is not an immutable library, so before every call the developer has to allocate the resulting matrix and pass it to the callee function. This is not a practical design choice, especially when we have to chain function calls on matrices of different sizes. Thus, the neural network utility functions built on top of this library adopted our functional approach.

Similar to Solidity, we mocked Java's random number generator to obtain the same results across all platforms.

Finally, the most noteworthy challenge of this part was the Emscripten compilation of C into executable WASM bytecode. In contrast to the revenue distribution compilation, we had to include the full precompiled GSL library in WASM during linking. This required to download all the source files of GSL and build them using the emmake and emconfigure tools of Emscripten. We then obtained a dynamic library in WASM as a shared object file which could be used in the linking process. Some extra source files still had to be included at linking to allow BLAS matrix multiplications.

3.3.3 Evaluation

We evaluated our neural network on the following arbitrary settings.

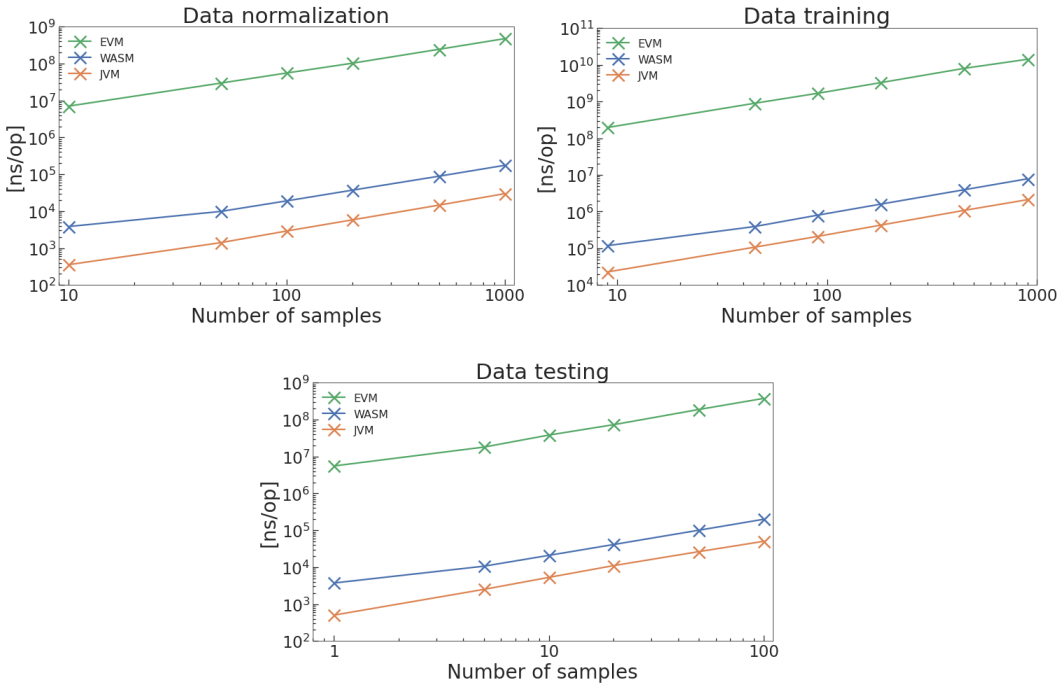
- 1 hidden layer of size 10
- 5 input features and 2 output classes
- 0.01 learning rate and 1 training epoch
- 90% of samples used for training, 10% for testing

This is a reduced but realistic configuration of a neural network. As we increased these parameters, we ran into problems when evaluating on the EVM, as we were reaching the maximum stack call depth of 1024.

We used dummy input data that could easily be generated across all platforms with a simple loop, as loading the MNIST dataset on Solidity was too constraining due to the absence of any I/O support.

Results

Following the configuration of **section 3.1.1** and the above mentioned neural network settings, we get the following results for data normalization, training and testing on our three different execution environments.



Once again, the EVM reports much worse results than the JVM and WASM. The slowdown ranges from circa **x10,000** to **x25,000** when comparing with the JVM.

This is an unsurprising result, since that in addition to the usual slow persistent memory operations, we also experience a slowdown due to the emulation of floating points and the continuous memory expansion caused by our immutable matrix library.

We illustrate this with a similar gas breakdown of a testing transaction involving 10 samples. We excluded opcodes that had less than a 5,000 total gas cost.

OPCODE	OCCURRENCES	TOTAL GAS
ADD	28,019	84,057
AND	15,176	45,528
CALLDATACOPY	110	6,833
DIV	3,021	15,105
DUP1	6,763	20,289
DUP2	25,096	75,288
DUP3	17,487	52,461
DUP4	12,238	36,714
DUP5	11,608	34,824
DUP6	4,823	14,469
DUP7	9,656	28,968
DUP8	3,045	9,135
DUP9	3,124	9,372
EQ	5,350	16,050
EXP	1,510	53,100
GT	3,718	11,154
ISZERO	17,988	53,964
JUMP	14,882	119,056
JUMPDEST	42,838	42,838
JUMPI	27,720	277,200
LT	11,430	34,290
MLOAD	12,672	38,016
MSTORE	3,642	30,984
MUL	7,371	36,855
NOT	5,048	15,144
OR	5,047	15,141
POP	34,188	68,376
PUSH1	96,575	289,725
PUSH2	51,061	153,183
SHL	23,760	71,280
SHR	9,751	29,253
SLOAD	1,692	1,353,600
SUB	14,795	44,385
SWAP1	32,502	97,506
SWAP2	8,762	26,286
SWAP3	13,099	39,297
SWAP4	3,177	9,531
TOTAL	**598,541**	**3,390,913**

Table 3.3: Gas for testing on 10 samples

We also reported a total of **2.95MB** memory footprint resulting from **16,485** memory expansions on this simulation.

As we can observe, the proportion of persistent memory operations only consists of around only **40%** of the total gas cost. The remaining **60%** is divided into about **16%** pure arithmetic operations, **12%** control flow and the balance of **32%** is volatile memory operations performing on the stack.

Thus, we show that a floating point emulation on Solidity, although convenient, comes with a high efficiency overhead. Finally, our memory footprint is not bounded by any garbage collector, explaining the high number of memory expansions.

We will end this section by detailing the reason behind the **x10** factor separating the JVM and WASM implementation. The slowdown is explained by the underlying inefficient matrix multiplication of the GSL library. Indeed, there is a distinction between CBLAS and BLAS matrix multiplications. Although BLAS is easier to use from a programming perspective, as it reuses the GSL matrix structures, CBLAS is the more optimized version that uses parallel SIMD instructions. Unfortunately, we only realized this problem too late and decided to keep this discrepancy as an illustration of C's programming burden.

Chapter 4

Conclusion

This chapter presents the general conclusions of our project and discusses its contribution to possible subsequent work.

4.1 Lessons learned

Overall, the objectives of this project have been met. We hope to have convinced the reader that the current status of deterministic smart contract execution on Ethereum remains limited.

We have shown through numerous examples the intrinsic limitations of Solidity and the EVM from a programming perspective on two showcase applications. We have highlighted a strong contrast between their ease of implementation on Solidity compared to their counterparts on generic programming languages such as Java and C.

We have demonstrated on these same numerically intensive applications the enormous slowdown occurring when running on the EVM, as opposed to other environments such as the JVM and WASM. While this slowdown can be explained by a variety of reasons, the absence of a built-in floating point support as well as an inefficient memory model contribute greatly to the observed slowdown.

On a more personal note, we realized along the way that C was probably not the ideal language for compiling to WASM. While we maintain that our initial reasons were laudable, we probably wasted too much time understanding the details of Emscripten. Unfortunately, this part made only a limited contribution to our project goal. Also, contrary to our initial belief, the WASM implementation did not outperform the JVM. We underestimated the efficiency overhead of translating from C to WASM and did not always make the right low-level choices to maximize our performance.

4.2 Next steps

We propose below a non-exhaustive list of extensions that can be brought to our work.

- **Open Vote Protocol**

  Originally planned as a third showcase application, we had to scale down our ambitions due to time constraints. The Open Vote protocol allows for fully decentralized anonymous elections on public blockchains. However, this requires extensive cryptographically intensive operations that are not natively supported by the EVM. Thus, reproducing our working methodology on this protocol would give us additional insight into the limitations of the EVM.[14]

- **Gas Metering**

  In this project, we used the author's personal hardware to evaluate the differences across the execution platforms. Thus, the times reported are not universally correct. Ethereum solves this problem by adding a universal gas cost to each opcode, which reflects the expected execution time on all computers. If a smart contract platform were to be created on top of WASM and the JVM, a similar metric should be developed.

- **Smart Contract Framework**

  As we elaborated in **section 2.2.1**, the current framework to develop smart contracts in Java and C is nonexistent. Therefore, we simulated the presence of some essential primitives such as the transaction sender or user address objects. The introduction of such an API can be done in parallel with the development of deterministic environments currently researched by the DEDIS lab.

4.3 Acknowledgments

Firstly, I would like to acknowledge my special gratitude to my responsible, Professor Bryan Ford, for allowing me to follow my path and develop this subject that I proposed personally.

Finally, I would like to thank my supervisor Enis Ceyhun Alp for his precious help and support throughout this semester.

Bibliography

[1] Enis Ceyhun Alp, Cristina Băsescu, Pasindu Tennage, Noémien Kocher, Gaylor Bosson, and Bryan Ford, *Efficient Deterministic Execution of Smart Contracts*, Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland

[2] Andreas M. Antonopoulous and Dr. Gavin Wood (2019), *Mastering Ethereum*, O'Reilly

[3] Dr. Gavin Wood, *Ethereum Yellow Paper*, `https://ethereum.github.io/yellowpaper/paper.pdf`

[4] Vitalik Buterin, *Ethereum: Platform Review*

[5] Gavin Zheng, Longxiang Gao, Liqun Huang, Jian Guan (2021), *Ethereum Smart Contract Development in Solidity*, Springer Nature Singapore Pte Ltd.

[6] Schlomer Alexis, *Project Repository*, `https://github.com/AlSchlo/BSc-Semester-Project`

[7] Guilio2002, *bsol source code*, `https://github.com/Giulio2002/bsol`

[8] Java Microbenchmark Harness (JMH) `https://github.com/openjdk/jmh`

[9] Schlomer Alexis (2022), *An Efficient Approach to Calculating the Cumulative Variable Revenue of Every Stakeholder in the Context of a Distributed Financial Application*, `https://github.com/AlSchlo/BSc-Semester-Project`

[10] Ethereum Official Website, *Patricia Merkle Trees*, `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/`

[11] tidwall, *hashmap source code*, `https://github.com/tidwall/hashmap.c`

[12] Zihan Zheng, Peichen Xie, Xian Zhang, Shuo Chen, et al. (2021), *Agatha: Smart Contract for DNN Computation*

[13] Odd Erik Gundersen, Saeid Shamsaliei, Richard Juul Isdahl, *Do machine learning platforms provide out-of-the-box reproducibility?*, Future Generation Computer Systems 126 (2022) 34–47

[14] Patrick McCorry, Siamak F. Shahandashti and Feng Hao, *A Smart Contract for Boardroom Voting with Maximum Voter Privacy*, School of Computing Science, Newcastle University UK

[15] ABDK-Consulting, *quad floating point numbers on Solidity*, `https://github.com/abdk-consulting/abdk-libraries-solidity`

[16] Michael Nielsen (2019), *Neural Networks and Deep Learning*, Online book: `http://neuralnetworksanddeeplearning.com/`

[17] Efficient Java Matrix Library (EJML), `http://ejml.org/wiki/index.php?title=Main_Page`

[18] GSL - GNU Scientific Library (GSL), `https://www.gnu.org/software/gsl/`

[19] The MNIST Database of handwritten digits, `http://yann.lecun.com/exdb/mnist/`