Diving Into The Ethereum VM Part 2 — How I Learned To Start Worrying And Count The Storage Cost

zh

Follow

--

8

Listen

Share

In the first article of this series we've peeked into the assembly code of a simple Solidity contract:

This contract boils down to an invocation of the sstore

instruction:

- The EVM stores the value 0x1

in the storage position 0x0

.

- Each storage position can store exactly 32 bytes (or 256 bits).

If this seems unfamiliar, I recommend reading:Diving Into The Ethereum VM Part 1 — Assembly & Bytecode

In this article we'll start to look into how Solidity uses chunks of 32 bytes to represent more complex data types like structs and arrays. We'll also see how storage could be optimized, and how optimization could fail.

In a typical programming language it's not terribly useful to understand how data types are represented at such a low-level. In Solidity (or any EVM language) this knowledge is crucial because storage access is super expensive:

- sstore

costs 20000 gas, or ~5000x more expensive than a basic arithmetic instruction.

- sload

costs 200 gas, or ~100x more expensive than a basic arithmetic instruction.

And by "cost", we are talking about real money here, not just milliseconds of performance. The cost of running & using your contract is likely to be dominated by sstore

and sload

!

Parsecs Upon Parsecs of Tape

It takes two essential ingredients to build an Universal Computation Machine:

1. A way to loop, either jump or recursion.
2. An infinite amount of memory.

The EVM assembly code has jump, and the EVM storage provides the infinite memory. That's enough for EVERYTHING, including simulating a world that runs a version of Ethereum, which is itself simulating a world that runs Ethereum that is…

The EVM storage for a contract is like an infinite ticker tape, and each slot of the tape holds 32 bytes. Like this:

We'll see how data lives on the infinite tape.

The length of the tape is 2^{256}, or ~10^{77} storage slots per contract. The number of particles of the observable universe is 10^{80}. About 1000 contracts would be enough to hold all those protons, neutrons, and electrons. Don't believe the marketing hype, as it is a lot shorter than infinity.

The Blank Tape

The storage is initially blank, defaulting to zero. It doesn't cost you anything to have an infinite tape.

Let's look at a simple contract to illustrate the zero-value behaviour:

The layout in storage is simple.

- The variable a

in position 0x0

- The variable b

in position 0x1

- And so on…

The key question: if we only use f

, how much do we pay for a, b, c, d, e?

Let's compile and see:

The assembly:

So a storage variable declaration doesn't cost anything, as there's no initialization necessary. Solidity reserves a position for that store variable, and you pay only when you store something in it.

In this case, we are only paying for storing to 0x5

.

If we were writing assembly by hand, we could choose any storage position without having to "expand" the storage:

Reading Zero

Not only can you write anywhere in storage, you can read from anywhere immediately. Reading from an uninitialized position simply returns 0x0

.

Let's see a contract that reads from a

, an uninitialized position:

Compile:

The assembly:

Notice that it's valid to generate code that sload

from an uninitialized position.

We can be smarter than the Solidity compiler, however. Since we know that tag_2

is the constructor, and a

had never been written to, we can replace the sload

sequence with 0x0

to save 5000 gas.

Representing Struct

Let's look at our first complex data type, a struct that has 6 fields:

The layout in storage is the same as state variables:

- The field t.a

in position 0x0

- The field t.b

in position 0x1

- And so on…

Like before, we can write directly to t.f

without paying for initialization.

Let's compile:

And we see the exact same assembly:

Fixed Length Array

Now let's declare a fixed length array:

Since the compiler knows exactly how many uint256 (32 bytes) there are, it can simply lay out the elements of the array in storage one after another, just as it did for store variables and structs.

In this contract, we are again storing to the position 0x5

.

Compile:

The assembly:

It is slight longer, but if you squint a little, you'd see that it's actually the same. Let's optimize this further by hand:

Removing the tags and spurious instructions, we arrive at the same bytecode sequence again:

Array Bound Checking

We've seen that fixed-length arrays has the same storage layout as struct and state variables, but the generated assembly code is different. The reason is that Solidity generates bound-checking for array access.

Let's compile the array contract again, turning off the optimizations this time:

The assembly is commented below, printing the machine state after each instruction:

We see the the bound-checking code now. We've seen that the compiler is able to optimize some of this stuff, but not perfectly.

Later in this article we will to see how array bound-checking interferes with compiler optimization, making fixed-length arrays much less efficient than store variables or structs.

Packing Behaviour

Storage is expensive (yayaya I've said it a million times). One key optimization is to pack as much data into one 32 bytes slot as possible.

Consider a contract that has four store variables, 64 bits each, adding up to 256 bits (32 bytes) altogether:

We'd expect (hope) that the compiler uses one sstore

to put these in the same storage slot.

Compile:

The assembly:

A lot of bit-shuffling that I can’t decipher, and I don’t care. The key thing to notice is that there’s only one sstore

.

Optimization success!

Breaking The Optimizer

If only the optimizer could work so well all the time. Let’s break it. The only change we make is that we use helper functions to set the store variables:

Compile:

The assembly output is too much. We’ll ignore most of the detail and focus on the structure:

Now there are now two sstore

instead of one. The Solidity compiler can optimize within a tag, but not across tags.

Calling functions could cost you far more, not so much because function calls are expensive (they are only jump instructions), but because sstore

optimization could fail.

To solve this problem, the Solidity compiler needs to learn how to inline functions, essentially getting the same code as not calling functions:

If we peruse the complete assembly output, we’d see that the assembly code for the functions

setAB()

and

setCD()

is included twice, bloating the size of the code, costing you extra gas to deploy the contract. We'll talk about this later when learning about the contract lifecycle.

Why The Optimizer Breaks

The optimizer doesn’t optimize across tags. Consider “1+1”, it can be optimized to 0x2

if under the same tag:

But not if the instructions are separated by tags:

This behaviour is true as of Version 0.4.13. Could change in the future.

Breaking The Optimizer, Again

Let’s see another way the optimizer fails. Does packing work for a fixed length array? Consider:

Again, there are four 64 bits numbers we’d hope to pack into one 32 bytes storage slot, using exactly one sstore

instruction.

The compiled assembly is too long. Let’s instead just count the number of sstore

and sload

instructions:

Oh noes. Even though this fixed-length array has exactly the same storage layout as equivalent struct or store variables, optimization fails. It now takes four pairs of sload

and sstore

.

A quick look at the assembly code reveals that each array access has bound checking code, and organized under different tags. But tag boundaries break optimization.

There is a small consolation though. The 3 extra sstore

instructions are cheaper than the first:

- sstore

costs 20000 gas for first write to a new position.

- sstore

costs 5000 gas for subsequent writes to an existing position.

So this particular optimization fail costs us 35k instead of 20k, 75% extra.

Conclusion

If the Solidity compiler can figure out the size of store variables, it simply lays them out one after another in storage. If possible, the compiler packs the data tightly in chunks of 32 bytes.

To summarize the packing behaviour we've seen so far:

- Store variables: yes.
- Struct fields: yes.
- Fixed-length arrays: no. In theory, yes.

Because storage access costs so much, you should think of the store variables as your database schema. When writing contracts, it could be useful to do small experiments, and examine the assembly to find out if the compiler is optimizing properly.

We can be sure that the Solidity compiler will improve in the future. For now, unfortunately, we can't trust its optimizer blindly.

It pays, literally, to understand your store variables.