Diving Into The Ethereum VM Part 3 — The Hidden Costs of Arrays

zh

Follow

--

3

Listen

Share

Solidity provides familiar data structures seen in other programming languages. Aside from simple values like numbers and structs, there are data types that can expand dynamically as more data is added. The three main categories of these dynamic types are:

- Mappings: mapping(bytes32 => uint256)

, mapping(address => string)

, etc.

- Arrays: []uint256

, []byte

, etc.

- Byte arrays. Only two kinds: string

, bytes

.

In Part II of this series we've seen how simpler types with fixed sizes are represented in storage.

- Fundamental values: uint256

, byte

, etc.

- Fixed sized arrays: [10]uint8

, [32]byte

, bytes32

- Structs that combine the above types.

Storage variables that have fixed sizes are laid out one after another in storage, packing as tightly as possible in chunks of 32 bytes.

(If this seems unfamiliar, read Diving Into The Ethereum VM Part II — Storage Cost)

In this article we'll look into how Solidity supports more complex data structures. Arrays and mappings in Solidity may look familiar on the surface, but the way they are implemented gives them radically different performance characteristics.

We'll start with mapping, which is the simplest of all three. It turns out that arrays and byte arrays are just mappings with fancier features.

Mapping

Let's store a single value in a uint256 => uint256

mapping:

Compile:

The assembly:

We can think of the EVM store as a key-value database, with each key limited to storing 32 bytes. Rather than using the key 0xC0FEFE

directly, here the key is hashed to 0x798...187c

, and the value 0x42

stored there. The hashing function used is the keccak256

(SHA256) function.

In this example we don’t see the keccak256

instruction itself because the optimizer had decided to precalculate the result and inline it in bytecode. We still see the vestige of this calculation, in the form of useless mstore

instructions.

Calculate The Address

Let’s use some Python code to hash 0xC0FEFE

to 0x798...187c

. If you'd like to follow along, you'll need Python 3.6, or installpysha3 to get the keccak_256

hash function.

Define two helper functions:

To convert numbers to 32 bytes:

To concatenate two byte arrays together, use the +

operator:

To calculate the keccak256 hash of some bytes:

We now have enough to calculate 0x798...187c

.

The position of the store variable items

is 0x0

(since it is the first store variable). To get the address, concatenate the key 0xc0fefe

with the position of items

:

The formula for calculating the storage address for a key is:

Two Mappings

Let’s take the formula we have to calculate where values would be stored! Suppose we have a contract with two mappings:

- The position for itemsA

position is 0

, for key 0xAAAA

:

- The position for itemsB

position is 1

, for key 0xBBBB

:

Let’s verify these calculations with the compiler:

The assembly:

As expected.

KECCAK256 in Assembly

The compiler was able to pre-calculate the address for a key because the values involved are constants. If the key used is a variable, then the hashing needs to be done with assembly code. Now we want to disable this optimization so we can see how hashing is done in assembly.

It turns out to be easy to cripple the optimizer, by introducing an extra indirection with a dummy variable i

:

The position for the variable items

is still 0x0

, so we should expect the same address as before.

Compile with optimization, but this time without the hash pre-calculation:

The assembly annotated:

The mstore

instruction writes 32 bytes in memory. The memory is much cheaper, costing only 3 gas to read and write. The first half of the assembly "concatenates" the key and position by loading them into neighbouring chunks of memory:

Then the keccak256

instruction hashes the data in that memory region. The cost depends on how much data is hashed:

- 30 Paid for each SHA3 operation.
- 6 Paid for each word of 32 bytes.

For an uint256

key the gas cost is 42 (30 + 6 * 2

).

Mapping Large Values

Each storage slot can only store 32 bytes. What happens if we try to store a struct that’s larger?

Compile, and you should see 3 sstore instructions:

Notice that the calculated addresses are the same except for the last digit. The member fields of the Tuple

struct are laid out one after another (..7d, ..7e, ..7f).

Mappings Don’t Pack

Given how mapping is designed, the minimal amount of storage you pay per item is 32 bytes, even if you are only storing 1 byte:

And if a value is larger than 32 bytes, you pay for storage in increments of 32 bytes.

Dynamic Arrays Are Mappings++

In a typical language, an array is just a list of items that sit together in memory. Say you have an array that contains 100 uint8

elements, then it would occupy 100 bytes of memory. In this scheme, it’s cheap to load the whole array in bulk onto CPU cache, and to loop through the items.

For most languages, arrays are cheaper than maps. For Solidity, though, array is a more expensive version of mapping. Items of an array would be laid out sequentially in storage, like:

But keep in mind, each access to these storage slots is in fact a key-value lookup in a database. Accessing an array element is no different from accessing a mapping element.

Consider the type []uint256

, it is essentially the same as mapping(uint256 => uint256)

with added features making it "array-like":

- length

to indicate how many items there are.

- Bound-checking. Throws error when reading & writing to an index larger than the length.
- More sophisticated storage packing behaviour than mapping.
- Automatic zeroing out of unused storage slots when an array is shrunk.
- Special optimization for bytes

and string

to make short arrays (less than 31 bytes) more storage efficient.

Simple Array

Let’s look at an array that stores three items:

The assembly code for array access is too complex to trace. Let’s use theRemix debugger to run the contract:

At the end of the simulation, we can see that 4 storage slots are used:

The position of the chunks

variable is 0x0

, which is used to store the array's length (0x3

). Hash the variable's position to find the address for storing array data:

Each item of the array is laid out sequentially from this address (0x29..63

, 0x29..64

, 0x29..65

).

Dynamic Array Packing

How about the all important packing behaviour? One advantage of array over mapping is that packing works. Four items of an uint128[]

array would fit in exactly two storage slots (plus 1 for storing the length).

Consider:

Run this in Remix, and the storage at the end looks like:

Only 3 slots are used, as expected. The length is again stored at 0x0

, the position of the storage variable. Four items are packed in two separate storage slots. The starting address for this array is the hash of the position of the variable:

The address now increments once for every two array elements. Looks good!

But the assembly code itself is not very well optimized. Since only two storage slots are used, we would hope that the optimizer uses two sstore

for the assignments. Unfortunately with bound-checking (and some other stuff) thrown in, it's not possible to optimize the sstore

instructions away.

Four sstore

instructions are used for the assignments:

Byte Arrays & String

bytes

and string

are special array types that optimize for bytes and characters respectively. If the length of the array is less than 31 bytes, only one storage slot is used to store the whole thing. Longer byte arrays are represented in much the same way as normal arrays.

Let’s see a short byte array in action:

Since the array is only 3 bytes (less than 31 bytes), it occupies just one storage slot. Run in Remix, the storage:

The data 0xaabbcc...

is stored from left to right. The 0

's that follow are empty data. The last byte 0x06

is the encoded length of the array. The formula is encodedLength / 2 = length

. In this case the actual length is6 / 2 = 3

.

A string works in exactly the same way.

A Long Byte Array

If the amount of data is greater than 31 bytes, a byte array is like []byte

. Let look at byte array that's 128 bytes long:

Run in Remix, and we see that four slots are used in storage:

The slot 0x0

is no longer used to store data. The whole slot now stores the encoded array length. To get the actual length, do length = (encodedLength - 1) / 2

. In this case the length is128 = (0x101 - 1) / 2

. The actual bytes are stored in0x290d...e563

, and the slots that follow sequentially.

The assembly code for byte array is quite big. Aside from the normal bound-checking and array resizing stuff, it also needs to encode/decode length, as well as taking care to convert between long and short byte arrays.

Why encode the length? Because the way it's done, there's a simple way to test if a byte array is short or long. Notice that the encoded length is always odd for a long array, and even for a short array. The assembly only needs to look at the last bit to see whether it is zero (even/short) or non-zero (odd/long).

Conclusion

Peeking into the inner workings of the Solidity compiler, we see that familiar data structures like mappings and arrays are radically different from conventional programming languages.

To recap:

- Arrays are like mappings, not very efficient.
- More complex assembly code than mappings.
- Better storage efficiency than mapping for smaller types (byte, uint8, string).
- Assembly not optimized very well. Even with packing, there's one sstore

per assignment.

The EVM storage is a key-value database, much like git. If you change anything, the checksum at the root node would change. If two root nodes have the same checksum, the stored data is guaranteed to be the same.

To appreciate how quirky Solidity & EVM is, imagine that each element of the array is its own file in a git repository. When you change the value of an array element, you are in fact creating a git commit. When iterating through an array, you can't load the whole array at once, you have to look into the repository and find each file separately.

Not only that, each file is limited to 32 bytes! Because we need to chop up data structures into 32 bytes chunks, Solidity's compiler is complicated by all kinds of logic and optimization tricks, all done in assembly.

Yet the 32 bytes limit is entirely arbitrary. The backing key-value store can store any number of bytes with a key. Perhaps in the future we could add a new EVM instruction to store arbitrary bytes with a key.

For now, the EVM storage is a key-value database prentending to be a 32 bytes array.

See ArrayUtils::resizeDynamicArray for a taste of what the compiler is up to when resizing an array. Normally data structures would be done in the language as part of a standard library, but in Solidity it's baked into the compiler.

If you enjoyed this article, you should follow me on Twitter

[@hayeah.

](https://twitter.com/hayeah)