SSZ stands for Simple SerialiZe.

You can read on some of the background and needs of SSZ from Raul Jordanhere.

Ethereum already has a serialization format called RLP (recursive length prefix).

SSZ allows for a few more things.

Deterministic lookup of an element in the bytes

You can look at the bytes of the SSZ output, and if you know what the data structure is, you can look up the bytes of one of the fields of the data structure directly.

Merkle hash partial computation

These are called SSZ partials. You can easily update the SSZ representation and recompute the hash without recomputing from scratch.

So SSZ has been meant for hashing in an efficient manner, offering optimizations. However we haven't really checked if those optimizations work well or help meaningfully at runtime, it's a spec after all right now.

Premature optimization is the root of all evil

OK so the whote text reads:

There is no doubt that the grail of efficiency leads to abuse.

Programmers waste enormous amounts of time thinking about,

or worrying about, the speed of noncritical parts of their programs,

and these attempts at efficiency actually have a strong

negative impact when debugging and maintenance are

considered. We should forget about small efficiencies,

say about 97% of the time: premature optimization is the root of all evil.

[

optimization_2x.png

569×840

](https://imgs.xkcd.com/comics/optimization_2x.png)

Pretty damning, but let's not forget the remaining 3%:

Yet we should not pass up our opportunities in that critical 3%.

A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.

After working with such tools for seven years, I've become convinced that all compilers written from now on should be designed to provide all programmers with feedback indicating what parts of their programs are costing the most; indeed, this feedback should be supplied automatically unless it has been specifically turned off.

Enter implementation.

The optimization implied by deterministic lookup is that you would be able to program so that it's possible to just keep the bytes of an object in SSZ encoding. It would also be able to use streaming.

In multiple occasions, folks who went down that route with ETH 1 over RLP mentioned there had been very little performance optimization using this method.

It actually makes implementation cumbersome and prone to breakage.

A few objects are encoded to SSZ with ETHv2. One of them is the block, and it's used in interesting ways:

- You have to use it in epoch processing to update it.
- You gossip it over the network to other peers.
- You store and index it locally.
- You receive blocks over the network and will need to verify them

It quickly becomes natural to have a in-memory representation of the object as a set of structs that can be tested, debugged, mocked and so on.

SSZ partials also suffer from complexity and constrain the use case. They require to keep a merkle tree around associated with the object, and each implementer team ends up creating quite a bit of code to support that use case.

Most likely for the longest time, until proven otherwise, implementers will rehash objects after updating them in their DSL so they can get a new hash for them, recreating the SSZ byte representation every time.

Lazy, lazy, lazy

Coders are lazy. Let's use it to our advantage and run here and now a thought experiment.

Take a business object of some sort, and use RLP to push it to bytes.

Why RLP? There is an implementation of RLP in every language out there already. It is used in prod by ETHv1. But you can do a mental switch to something mature and used out there today, as long as it is absolutely deterministic.

So you get those RLP bytes, now let's stream those bytes into a zip compression stream. Same as before - use whatever mean is your favorite, with those requirements: compression must be implemented in all languages, deterministic and if possible so common it's hardware accelerated.

Finally, compute a hash of the zip stream using SHA256. Again, same requirements.

You get a hash for your object. You streamed so you hopefully didn't clog heap space with bytes. You have a reasonably fast throughput so you can redo this on a whim.

Is this even efficient?

Well two ways:

It simplifies concepts

I quote Ben Edgington: "simplicity is not just about lines of code, it is primarily about the concepts we are implementing."

It helps dev stay lazy

This liberates head space so we can move on to bigger and better things, and lowers the bar for contribution.