Formally-verified optimised epoch processing

By Callum Bannister and Michael Sproul, Sigma Prime

.

Implementations of the Ethereum consensus specification typically make use of algorithmic optimisations to achieve high performance. The correctness of these optimisations is critical to Ethereum's security, and so far they have been checked by manual review, testing and fuzzing. To further increase assurance we are formally proving the correctness of an optimised implementation.

This document describes the optimised implementation which we are in the process of verifying, and includes a high-level argument for its correctness.

Scope

We only consider the process_epoch function which is responsible for computing state changes at the end of each 32-slot epoch. Block processing (process_block) is considered out of scope for now but may be covered by future work.

Our goal is to verify an implementation of process_epoch containing the minimum number of O(n) iterations over the validator set (n is the number of validators). We consider not only the state.validators field, but also the other length n fields of the state including:

- .validators
- .balances
- .previous_epoch_participation
- .current_epoch_participation
- .inactivity_scores

The specification version targeted is v1.3.0, for the Capella hard fork. We anticipate that the proofs will be able to be updated for Deneb quite easily because there are minimal changes to epoch processing in the Deneb fork.

Motivation

As the validator set grows the amount of computation required to process blocks and states increases. If the algorithms from consensus-specs were to be used as-is, the running time of process_epoch would be increasing quadratically (O(n^2)) as validators are added.

Another motivation for optimising epoch processing is that it grants implementations the freedom to explore different state models. Some clients have already switched their BeaconState representation from an array-based model to a tree-based model, which allows for better sharing of data between states, and therefore better caching. The downside of the tree-based model is that it tends to have substantially slower indexing (e.g. computing state.validators[i]), and iteration is slightly slower (same time complexity with a larger constant).

Operation

Array-based

Tree-based

index

O(1)

O(\log n)

iterate

O(n)

O(c * n)

Hence in the tree-based paradigm it becomes even more important to remove random-access indexing,

and to remove O(n^2)

nested iterations which amplify the higher cost of tree-based iteration.

Algorithm Description

For ease of keeping this post up-to-date, we link to the algorithm description in our main Git repository:

- [Algorithm Description

](https://github.com/sigp/verified-consensus/blob/e71cde15e5f42dce6aa0eac5093945deb1027930/docs/description_and_informal_proof.md#algorithm-description) @ milestone1

tag; Nov 2023.

- [Algorithm Description

](https://github.com/sigp/verified-consensus/blob/e71cde15e5f42dce6aa0eac5093945deb1027930/docs/description_and_informal_proof.md#algorithm-description) @ main

branch; current.

Informal Proof Sketch

- [Informal Proof Sketch

](https://github.com/sigp/verified-consensus/blob/e71cde15e5f42dce6aa0eac5093945deb1027930/docs/description_and_informal_proof.md#informal-proof-sketch) @ milestone1

tag; Nov 2023.

- [Informal Proof Sketch

](https://github.com/sigp/verified-consensus/blob/e71cde15e5f42dce6aa0eac5093945deb1027930/docs/description_and_informal_proof.md#informal-proof-sketch) @ main

branch; current.

Separation Logic Algebra

As part of this work we've developed an Isabelle/HOL theory for verifying the correctness of the optimised implementation (relative to the original).

It combines several layers in a novel way

- We use the Concurrent Refinement Algebra

(CRA) developed by Hayes et al as the unifying language for the formal specification and refinement proof between the original and optimised implementation.

- We implement a concrete semantics

of said algebra using an intermediate

model of Order Ideals as bridge between CRA and a trace semantics.

- We denote programs using the Continuation Monad (roughly mimicking a Nondeterministic State Monad with failure) to provide a familiar Haskell-style syntax and simulate argument-passing in the CRA.

- We extend the notion of ordinary refinement in CRA to data refinement

.

- Finally, we use Separation Logic as an assertion language

, allowing reasoning about the spatial independence

of operations as required for the optimised implementation to preserve the original semantics.

At the time of writing the framework is mostly complete but has a few proofs skipped (using Isabelle's sorry

) which we intend to revisit later.

Links Below

- [Separation Logic Algebra

](https://github.com/sigp/verified-consensus/tree/milestone1/algebra) @ milestone1

tag; Nov 2023.

- [Separation Logic Algebra

](https://github.com/sigp/verified-consensus/tree/main/algebra) @ main

branch; current.

Implementation and Fuzzing

We have implemented the optimised algorithm on Lighthouse's tree-states

branch, which uses tree-based states and benefits significantly from the reduction in validator set iteration. The Lighthouse implementation closely follows the described algorithm, with some minor variations in the structure of caches, and some accommodations for Deneb which we argue are inconsequential.

The Lighthouse implementation is passing all spec tests as of v1.4.0-beta.2.

- [single_pass.rs

]
(https://github.com/sigp/lighthouse/blob/d36ebba1eaf4e337fb0038691f2f74ff953b12c7/consensus/state_processing/src/per_epoch_processing/single_pass.rs): bulk of the Lighthouse implementation; Rust.

- [GitHub actions for ef-tests

](https://github.com/sigp/lighthouse/actions/runs/6568484722/job/17842965682): Successful CI run for the Ethereum Foundation spec tests on the tree-states

branch.

The Lighthouse implementation is also currently undergoing differential fuzzing against the other clients, as part of the beaconfuzz

project. So far no bugs have been discovered.

Next Steps

The next step is to formalise both the spec and our implementation in the separation logic framework within Isabelle/HOL, and then prove refinement following the proof sketches.

1. Port the partially-written spec code from the option

monad to the new continuation monad.

1. Translate the optimised algorithm to Isabelle/HOL code following the Python algorithm description.
2. Prove refinement proceeding through the phases of epoch processing in order. Starting from process_justification_and_finalization_fast

and building out supporting auxiliary lemmas as we go. The proof sketches provide high-level guidance for this step.

In parallel with the above we will also continue fleshing out the logical framework, and completing the proofs.

We plan to have this work completed by Q2 2024.

Acknowledgements

We'd like to thank the Ethereum Foundation for a grant supporting this research, and Sigma Prime for facilitating the project.