Scalable Reward Distribution with Compounding Stakes

Richard Pardoe *

December 2020
v1.0

Abstract

Scalability of Ethereum dApps is constrained by the gas costs of writing to storage. In previous work, Batog [1] and Solmaz [2] solved the problem of scalable reward distribution in a staking pool, distributing rewards in proportion to a staker's share of total stakes with a "pull" based approach. Only a single running sum need be updated at each reward, sidestepping the need for an expensive $O(n)$ storage update for n stakers. However, their work stopped short of solving the problem for a compounding stake. Here we derive formulas for a scalable implementation of a compounding, decreasing Stability Pool deposit and its corresponding ETH gain. Thus, Liquity tracks the deposits and rewards of every depositor with only $O(1)$ complexity for each liquidation.

1 Introduction

The Liquity protocol uses a Stability Pool as its primary mechanism to liquidate collateralized debt positions (called "troves") that fall below the minimum collateralization ratio of 110% [3]. Stability deposits (provided in LUSD) absorb and cancel the debt (in LUSD) from defaulted troves: an amount of LUSD in the Stability Pool corresponding to the debt of the liquidated trove is burned from the pool's balance, offsetting the debt. In return, Stability Pool depositors are rewarded with the acquisition of collateral (in ETH) from liquidated positions at a significant discount. The proportion of a stability depositor's current deposit to the total LUSD in the pool determines the collateral share it receives from the liquidation.

The fact that liquidated troves decrease all deposits in the Stability Pool leads to an implementation challenge: Due to Ethereum's block gas limit, it becomes impossible to iterate over all deposits in the Stability Pool for a large number of deposits. Such an operation has computational complexity of $O(n)$ and does not scale.

This paper presents a method to perform liquidations against the Stability Pool in a scalable manner with only $O(1)$ complexity. The method is "pull-based": Instead of adjusting all stability deposits upon every liquidation, the update is deferred to the moment at which an owner withdraws or tops up their deposit.

The methods presented in this paper are generalizable and can be used for other dApps running on Ethereum.

2 Derivation

For a pull-based implementation in Liquity's Stability Pool, we must factor out the initial deposit, and track reward terms that are independent from the individual deposit values. We derive formulas for a compounding LUSD deposit and corresponding ETH gain as functions of the initial deposit, and involving a product P and a sum S: the former is a function of the LUSD losses, and the latter is a function of P, the ETH rewards, and the total deposits.

*rick@liquity.org

Terms

d_i : A given user's LUSD deposit at liquidation event i

Q_i : The LUSD debt absorbed by the Stability Pool from liquidation i

E_i : The ETH sent to the Stability Pool from liquidation i

D_i : Total LUSD deposits at liquidation i

Let d_0 represent the user's initial Stability Pool deposit. After the first liquidation, their new deposit d_0 is given by:

$$d_1 = d_0 - LUSDLoss \tag{1}$$

Liquidation assigns an LUSD loss to the deposit proportional to the prior share of total deposits:

$$d_1 = d_0 - Q_1 \frac{d_0}{D_0} \tag{2}$$

$$d_1 = d_0 \left(1 - \frac{Q_1}{D_0}\right) \tag{3}$$

Since the deposit compounds, the deposit value after liquidation 2 is a function of the previous deposit value:

$$d_2 = d_1 \left(1 - \frac{Q_2}{D_1}\right) \tag{4}$$

and substituting equation (2):

$$d_2 = d_0 \left(1 - \frac{Q_2}{D_1}\right) \left(1 - \frac{Q_1}{D_o}\right) \tag{5}$$

Similarly:

$$d_3 = d_2 \left(1 - \frac{Q_3}{D_2}\right) \tag{6}$$

$$d_3 = d_0 \left(1 - \frac{Q_3}{D_2}\right) \left(1 - \frac{Q_2}{D_1}\right) \left(1 - \frac{Q_1}{D_0}\right) \tag{7}$$

And the general case for a compounded deposit:

$$d_n = d_0 \prod_{i=1}^{n} \left(1 - \frac{Q_i}{D_{i-1}}\right) \tag{8}$$

2.1 Compounded deposit for initial deposit made at $t > 0$

For a deposit made between liquidations $[t, t+1]$, which is withdrawn between liquidations $[s, s+1]$, rewards are earned from liquidations $t+1, t+2, \ldots, s$.
We take a snapshot of the product at liquidation t, and calculate the compounded stake at liquidation n, with $t < n \leq s$, as:

$$d_n = d_t \prod_{i=t+1}^{n} \left(1 - \frac{Q_i}{D_{i-1}}\right) = d_t \frac{\prod_{i=1}^{n} \left(1 - \frac{Q_i}{D_{i-1}}\right)}{\prod_{i=1}^{t} \left(1 - \frac{Q_i}{D_{i-1}}\right)} \tag{9}$$

This assumes that every multiplicand is non-zero, i.e. that $Q_i < D_{i-1}$ for all i (see section 3.1 for the case where $Q_i = D_{i-1}$ and the pool is emptied).

Labelling the product term P_k, i.e: $P_k = \prod_{i=1}^{k} \left(1 - \frac{Q_i}{D_{i-1}}\right)$, yields:

$$d_n = d_t \frac{P_n}{P_t} \tag{10}$$

Where P_n is the product at liquidation n, and P_t is the snapshot of the product at the time when the user made the deposit.

2.2 Corresponding ETH Gain

The LUSD deposit earns an ETH gain at each liquidation. At each liquidation i, the user's LUSD deposit effectively decreases. Thus we can write the user's cumulative ETH gain e_n, from a series of n liquidations, as:

$$e_n = d_0 \left(\frac{E_1}{D_0}\right) + d_1 \left(\frac{E_2}{D_1}\right) + d_2 \left(\frac{E_3}{D_2}\right) + ... + d_{n-1} \left(\frac{E_n}{D_{n-1}}\right) \tag{11}$$

Then, using our expression for the deposit from equation (8):

$$e_n = d_0 \left(\frac{E_1}{D_0}\right) + d_0 \left(\frac{E_2}{D_1}\right) \prod_{i=1}^{1} \left(1 - \frac{Q_i}{D_{i-1}}\right) + d_0 \left(\frac{E_3}{D_2}\right) \prod_{i=1}^{2} \left(1 - \frac{Q_i}{D_{i-1}}\right) + ... + d_0 \left(\frac{E_n}{D_{n-1}}\right) \prod_{i=1}^{n-1} \left(1 - \frac{Q_i}{D_{i-1}}\right) \tag{12}$$

And factoring out the initial deposit, d_0:

$$e_n = d_0 \sum_{k=1}^{n} \left[\frac{E_k}{D_{k-1}} \prod_{i=1}^{k-1} \left(1 - \frac{Q_i}{D_{i-1}}\right) \right] = d_0 \sum_{k=1}^{n} \left[\frac{E_k}{D_{k-1}} P_{k-1} \right] \tag{13}$$

Where the initial product term $\prod_{i=1}^{0}(...) = 1$.

Thus, we have the depositor's cumulative ETH gain e_n as a function of the initial deposit d_0, and the rewards and total deposits at each liquidation.

Let the summation $\sum_{k=1}^{n}[...]$ be denoted S_n.

2.3 ETH gain for initial deposit made at $t > 0$

For a deposit made between liquidations $[t, t+1]$, which is withdrawn between liquidations $[s, s+1]$, rewards are earned from liquidations $t+1, t+2, \ldots, s$.
We take a snapshot of the product at liquidation t, and calculate the compounded stake at liquidation n, with $t < n \leq s$, as:

To account for the deposit entering after liquidations have already begun, we correct the product terms in (13) to $\frac{P_n}{P_t}$, as per (10):

$$e_n = d_t \sum_{k=t+1}^{n} \left[\frac{E_k}{D_{k-1}} \frac{P_{k-1}}{P_t} \right] \tag{14}$$

We take a snapshot of the sum at liquidation t, S_t. Then, the ETH gain earned by the deposit at liquidation n is:

$$e_n = d_t \frac{S_n - S_t}{P_t} \tag{15}$$

3 Basic Implementation

Making a deposit:
Record deposit: deposit[user] $= d_t$
Update total deposits: $D = D + d_t$
Record product snapshot: $P_t = P$
Record sum snapshot: $S_t = S$

Upon each liquidation yielding LUSD debt 'Q' offset with the Stability Pool and ETH gain 'E':
Update S: $S = S + \frac{E}{D}P$ (*intuition: a deposit's marginal ETH gain is equal to the deposit* $\times$ *ETH per unit staked* $\times$ *current correction factor)*
Update P: $P = P(1 - \frac{Q}{D})$
Update total deposits: $D = D - Q$

Withdrawing the deposit and ETH gain:
Compute final compounded LUSD deposit d: $d = d_t \frac{P}{P_t}$
Compute final corresponding cumulative ETH gain e: $e = d_t \frac{S - S_t}{P_t}$
Send d and e to user
Update deposit: deposit[user] $= 0$
Update total deposits: $D = D - d$

4 Practical Implementation in Liquity

Two further considerations are needed for our implementation:

1. Liquidations that completely empty the Stability Pool
2. How to handle the eternally decreasing product P, without truncating to 0

4.1 Liquidations that completely empty the Pool

Problem: Pool-emptying should reduce all deposits to 0, but we should not set P to 0. Doing so would break deposit computations for all future deposits: as P is a running product, all snapshots and future values would be 0.

Solution: Complete pool-emptying is handled by tracking a "current epoch" variable. The ETH gain reward sum S for each epoch is stored in a mapping.

Upon a pool-emptying liquidation, S is first updated as usual, for the current epoch. Then, the current epoch is incremented by 1, and P and S terms are reset for the new epoch. By definition, deposits that were made in past epochs will have been completely cancelled with liquidated debt, and so reduced to 0.

Making a deposit:
Record a snapshot of the current epoch for the deposit

Liquidation that empties the Pool:
Compute the latest value of S, and store it for the current epoch. Increase the epoch by 1, and reset the product and sum ($P = 1$, $S = 0$).

Withdrawing Deposit:
When users withdraw, check their epoch snapshot against the current epoch: If equal, deposit and ETH gain are computed as normal.

If the current epoch is greater than the deposit's epoch snapshot, then the deposit was made before the pool-emptying liquidation. Therefore:

- The **compounded deposit** is 0, as the deposit has been fully used to absorb debt.
- The user's **ETH gain** is computed using the S sum that corresponds to the epoch in which their deposit was made.

4.2 Eternally decreasing the product P, without truncating to 0

At liquidation i, P is multiplied by some new term $0 < p_i < 1$. Thus in theory, P is always decreasing, but should never reach 0.

Problem: We cannot represent an arbitrarily small value in Solidity. Eventually, division will truncate a small value to 0. If so, all future values of P would be 0, and deposit computation would break.

Solution: use a "current scale" that allows P to decrease indefinitely, but never reach 0. Each scale represents a division by 1e18.

Upon a liquidation that would otherwise truncate P to 0, S is first updated as usual, for the current scale. Then, the current scale is incremented by 1, and P is updated and scaled by 1e18.

Making a deposit:
Record a snapshot of the current scale for the deposit

Liquidation:
Compute the latest value of S, and store it for the current scale. Liquidation L_i causes the product P to be multiplied by some new product factor p_i. If $P \times p_i$ would be truncated to 0, instead do: $P = (P \times p_i \times 1e18)$, and increment the current scale by 1.

Withdrawing a deposit:
First, compute the number of scale changes made during the deposit's lifetime, i.e. (currentScale - scaleSnapshot).

If the number of scale changes is 0, compute the deposit and ETH gain as normal.

Otherwise:

Compounded deposit: If a scale change in P was made during the deposit's lifetime, account for it in the deposit computation. If the deposit has decreased by a factor of $<$ 1e-18 (i.e. if more than one scale change was made) just return 0.

ETH Gain: Since the deposit may span up to one scale change, so too does the reward. In this case, obtain the ETH gain using the S sums from the two consecutive scales that the deposit spans. The reward from the second sum is scaled by 1e-18. The rewards from both scales are added to make the final ETH gain.

5 Putting it all together

The implementation relies on a nested mapping: epochToScaleToSum.

The inner mapping stores the sum S at different scales, for a given epoch. The outer mapping stores the $(scale \Rightarrow sum)$ mappings for each epoch.

This allows us to track the ETH reward terms at each scale and epoch.Thus we can correctly compute the ETH rewards for a compounding decreasing stake, taking account of periodic Pool-emptying, and getting around the limitations of Solidity arithmetic.

All implementation logic above is combined in PoolManager.sol, in the functions:

$\_getCompoundedLUSDDeposit()$
$\_getDepositorETHGain()$
$\_updateRewardSumAndProduct()$

6 Acknowledgements

We would like to thank Daniel Simon, Robert Lauko, Bingen Eguzkitza and Onur Solmaz for the helpful discussion on reward distributions and Bojan Peček for his help with preparing this paper.

References

[1] B. Batog, L. Boca, N. Johnson. *Scalable Reward Distribution on the Ethereum Blockchain, 2018.* http://batog.info/papers/scalable-reward-distribution.pdf

[2] O. Solmaz. *Scalable Reward Distribution with Changing Stake Sizes", 2019.* https://solmaz.io/2019/02/24/scalable-reward-changing/

[3] R. Lauko, R. Pardoe. *Liquity: Decentralized Borrowing Protocol (Whitepaper), 2020.* https://docsend.com/view/bwiczmy