An $O(1)$ Dynamic Reward Rate Distribution Algorithm for Smart Contracts

Ivan Menshchikov [*]

juglipaff@gmail.com

April 15, 2023

Abstract

This paper presents an algorithm for efficient decentralized distribution of rewards among pool participants with a variable reward rate based on the amount of reward tokens to be distributed, which achieves $O(1)$ time complexity. The proposed algorithm is particularly useful for yield farming projects on Ethereum blockchain, where gas constraints make it impractical to distribute rewards at every block. The paper demonstrates the efficiency and scalability of the proposed algorithm and presents some potential use cases for its application in smart contracts.

1. Introduction

Decentralized finance (DeFi) has emerged as one of the most exciting and rapidly growing sectors in the blockchain ecosystem. DeFi protocols enable users to perform financial transactions such as lending, borrowing, trading, and staking in a decentralized and trustless manner. One of the core features of DeFi protocols is the ability to distribute rewards to participants in exchange for their contribution to the network, such as providing liquidity, staking tokens [1], or performing governance functions. [2]

Reward distribution is a critical component of many DeFi protocols, and its efficient and fair implementation is essential for the sustainability and growth of these projects. However, designing an optimal reward distribution algorithm is not a trivial task, and several challenges need to be addressed, including gas constraints, varying reward rates, and the need to scale to a large number of participants.

Several algorithms have been proposed to address these challenges, but most of them suffer from significant limitations. For example, *Scalable Reward Distribution on the Ethereum Blockchain* (Batog, Boca & Johnson, 2018) [3] is only suitable for very fine grained distribution (e.g. every block), which is not feasible for most projects due to gas constraints. Synthetix's StakingRewards [4] contract and Sushi's MasterChief [5] both assume constant time interval or reward rate, which does not work for yield farming projects with variable token rewards.

In this paper, I propose a novel algorithm for decentralized reward distribution in smart contracts that overcomes these limitations while retaining $O(1)$ time complexity. My algorithm can handle variable reward rates based on the amount of reward tokens needed to be distributed and is particularly useful for yield farming projects that have long time intervals between token distributions. I also discuss the implementation of my algorithm as an Ethereum smart contract and also compare my approach with other existing algorithms and highlight it's advantages.

2. Related Work

Reward distribution is a well-studied problem in the context of blockchain-based systems, and several algorithms have been proposed to address it. In this section, I provide an overview of the related work on reward distribution in smart contracts.

[*] Special thanks to Roman Vinogradov and Anton Dementyev

2.1. Constant reward rate algorithms

Several DeFi protocols use constant reward rate algorithms for reward distribution. For example, Sushiswap's MasterChief [5] contract assumes a constant reward rate, which means that the rate of token rewards is fixed and does not change over time. Although constant reward rate algorithms are simple and easy to implement, they do not work well for yield farming projects with variable token rewards.

2.2. Time-based algorithms

Another class of reward distribution algorithms is based on time intervals. These algorithms distribute rewards at fixed time intervals, such as daily or weekly. However, time-based algorithms suffer from several limitations, such as the inability to handle variable time intervals, the need for manual intervention to adjust the distribution frequency, and the potential for reward accumulation and loss due to missed intervals. As an example, Synthetix's StakingRewards [4] algorithm belongs to this category.

2.3. Off-chain algorithms

Off-chain distribution algorithms, such as those using Merkle trees [6][7], have become popular for efficient reward distribution. They store reward distribution information off-chain and use cryptographic proofs for verification, reducing gas costs and increasing security. However, they may require additional infrastructure and coordination between off-chain and on-chain components. Also if the off-chain storage and calculation of such an algorithm is controlled by a single entity e.g. centralized server, then it would be a centralized solution.

2.4. Block-based algorithms

Some reward distribution algorithms are based on assumption that the rewards are distributed every time a new block is added to the blockchain. While block-based algorithms can provide scalable and optimal reward distribution, they suffer from gas constraints and are not a practical solution for most DeFi projects. An example of such an algorithm is *Scalable Reward Distribution on the Ethereum Blockchain* (Batog, Boca & Johnson, 2018) [3]. This algorithm does not track the time when participants enter the pool and therefore does not work well with long distribution intervals. Participants could join the pool just before reward distribution and receive rewards they did not earn.

2.5. My solution

While several algorithms have been developed to handle reward distribution in smart contracts, most of them suffer from significant limitations. My proposed algorithm overcomes these limitations and achieves $O(1)$ time complexity, making it an efficient and scalable solution for reward distribution in DeFi protocols.

3. Reward Distribution Algorithm

I start by outlining the key components of my algorithm:

- `Deposit(amount)` : This function allows participants to deposit their tokens into the pool. The function updates the participant's stake and the total amount of tokens deposited.
- `Withdraw(amount)` : This function allows participants to withdraw amount of tokens from their stake. The function updates the participant's stake and the total amount of tokens remaining in the distribution pool.

- `CollectReward(amount)` : This function allows participants to collect their reward from the pool. The function calculates the participant's reward based on their stake, the time their tokens have been in the distribution pool and the distributions the participant was part of.

- `Distribute(reward)` : This function distributes rewards between participants. The contract implements a pull-based system to keep constant time complexity. Therefore, participants have to call `CollectReward` afterwards to collect their rewards.

The algorithm also needs to track the times the participants made their deposits to avoid the issue I described at section 2.4. There are a number of challenges that make calculating rewards a participant should receive non-trivial:

1. A participant can have an unlimited number of deposits and withdrawals at different blocks with different amounts.

2. An unlimited number of distributions with different reward amounts can happen at different time intervals.

3.1. Participant's reward after a single distribution

Let's consider the chronological order of all the deposit, withdrawal and distribution events at a single distribution. Let T_1, T_2, ..., T_n be the sum of all active stakes at each block t_1, t_2, ..., t_n in which deposit or withdrawal took place, where n is the block at which *reward* is distributed. A participant i would have $stake_{i,1}$, $stake_{i,2}$, ..., $stake_{i,n}$ at each such block depending on their deposits and withdrawals.

Let's name the following expressions:

$$depositAge_i = \sum_{k=1}^{n} (stake_{i,k} * (t_k - t_{k-1})) \tag{1}$$

$$totalDepositAge = \sum_{k=1}^{n} (T_k * (t_k - t_{k-1})) \tag{2}$$

Participant i will receive a reward of:

$$r_i = \frac{depositAge_i}{totalDepositAge} * reward \tag{3}$$

We can accumulate *totalDepositAge* on each deposit and withdrawal, making it possible to compute (2) in constant time on i's collection of rewards. Because the stake of i is constant over the interval in which no deposits or withdrawals were made by them, we can rewrite (1) as:

$$depositAge_i = \sum_{p=1}^{L_i} (stake_{i,A_{i,p}} * (t_{A_{i,p}} - t_{A_{i,p-1}})) \tag{4}$$

$$If\ p = 0,\ A(p) = 0 \tag{5}$$

Where L_i is the amount of deposits and withdrawals made by participant i and $A_{i,p}$ is the block i made action p. It is also possible to accumulate $depositAge_i$ for participants on their deposits and withdrawals allowing us to compute (4) in constant time. Therefore, it is also possible to compute r_i in constant time.

3.2. Participant's reward after multiple distributions

Let's consider the case in which participant i deposits their stake before distribution d and holds it for multiple distributions $d, d+1, ..., D$. Their reward R would be:

$$R_i = \sum_{m=d}^{D} (\frac{depositAge_{m,i}}{totalDepositAge_m} * reward_m) \tag{6}$$

Participant i will receive a reward of (3) after the first distribution d. If $m > d$ their stake would remain constant:

$$depositAge_{m,i} = stake_{i,A_{i,L_i}} * (t_{m,n_m} - t_{m-1,n_{m-1}}) \tag{7}$$

Where $stake_{i,A_{i,L_i}}$ is the stake of i after their last action which has happened in the distribution interval d and $t_{m,n_m} - t_{m-1,n_{m-1}}$ is the length of distribution interval m in blocks.

$$R_i = r_{d,i} + stake_{i,A_{i,L_i}} * \sum_{m=d+1}^{D} (\frac{t_{m,n_m} - t_{m-1,n_{m-1}}}{totalDepositAge_m} * reward_m) \tag{8}$$

We can rewrite (8) the following way:

$$R_i = r_{d,i} + stake_{i,A_{i,L_i}} * (\sum_{m=1}^{D} (\frac{t_{m,n_m} - t_{m-1,n_{m-1}}}{totalDepositAge_m} * reward_m) - \sum_{m=1}^{d} (\frac{t_{m,n_m} - t_{m-1,n_{m-1}}}{totalDepositAge_m} * reward_m)) \tag{9}$$

We can accumulate this sum and store it for each distribution d:

$$rewardAgePerDepositAge_d = \sum_{m=1}^{d} (\frac{t_{m,n_m} - t_{m-1,n_{m-1}}}{totalDepositAge_m} * reward_m) \tag{10}$$

So (9) becomes:

$$R_i = r_{d,i} + stake_{i,A_{i,L_i}} * (rewardAgePerDepositAge_D - rewardAgePerDepositAge_d) \tag{11}$$

We can store the last distribution D on each new distribution and store the next distribution d for each participant i on each their action making it possible to calculate $rewardAgePerDepositAge_D - rewardAgePerDepositAge_d$ in constant time.

$r_{d,i}$ can also be calculated in constant time by storing $\frac{reward_d}{totalDepositAge_d}$ for each distribution d and multiplying it by $depositAge_i$ on i's collection of rewards. Therefore it is also possible to compute R_i in constant time.

3.3. Other cases

Every other case can be modeled by adding multiple (11) expressions together and substituting d for the next distribution after i's last action. The algorithm remains constant time because we can accumulate i's rewards on each of their action.

3.4. Algorithm

Algorithm: An $O(1)$ Dynamic Reward Rate Distribution Algorithm

```
function Initialization(block):
    ID = 0; // Store the ID of the last distribution
    d[ID].block = block; // Store initialization block in the distribution data
    lastUpdate = block; // Last update block
    T = 0; // Total deposits
    totalDA = 0; // Total deposit age

function Deposit(block, i, amount) public:
    updateDepositAge(block, i); // Update on each action
    participant[i].stake += amount;
    T += amount;

function Withdraw(block, i, amount) public:
    updateDepositAge(block, i); // Update on each action
    if amount > participant[i].stake then
        revert(); // Not enough balance
    participant[i].stake -= amount;
    T -= amount;

function CollectReward(block, i, amount) public:
    updateDepositAge(block, i); // Update on each action
    if amount > participant[i].reward then
        revert(); // Not enough reward balance
    participant[i].reward -= amount;

function Distribute(block, reward) public:
    if T == 0 then
        revert();
    if d[ID].block == block then
        revert();
    // Add remaining deposit age and calculate reward per total deposit age
    uint rewardPerDA = reward / (totalDA + T * (block - lastUpdate));
    // Calculate reward age per total deposit age and add it to previous sumRewardAgePerDA
    uint sumRewardAgePerDA = d[ID].sumRewardAgePerDA + rewardPerDA * (block - d[ID].block);

    ID += 1;
    d[ID] = {
        block: block,
        rewardPerDA: rewardPerDA,
        sumRewardAgePerDA: sumRewardAgePerDA
    };
    lastUpdate = block;
    totalDA = 0;
```

Algorithm: An $O(1)$ Dynamic Reward Rate Distribution Algorithm

```
function UpdateDepositAge(block, i) internal:
    if participant[i].nextID == ID + 1 then
        // If the distribution did not happen after participant[i].lastUpdate we accumulate i's deposit age
        participant[i].DA += participant[i].stake * (block - participant[i].lastUpdate);
    else
        // If the distribution has happened after participant[i].lastUpdate we update i's reward and start
           accumulating i's deposit age from zero
        participant[i].reward = Reward(i);
        participant[i].DA = participant[i].stake * (block - d[ID].block);
    participant[i].nextID = ID + 1;
    participant[i].lastUpdate = block;
    totalDA += (block - lastUpdate) * T; // Accumulate total deposit age
    lastUpdate = block;

function Reward(i) public:
    if participant[i].nextID == ID + 1 then
        // If the distribution after i's last deposit did not yet happen
        return participant[i].reward;
    // Add remaining deposit age and calculate reward between i's last update and the distribution after
    uint DA = participant[i].DA + participant[i].stake * (d[participant[i].nextID].block - participant[i].lastUpdate);
    uint rewardBeforeD = DA * d[participant[i].nextID].rewardPerDA;
    // Calculate reward from the distributions that have happened after the last user deposit
    uint deltaRewardAgePerDA = d[ID].sumRewardAgePerDA - d[participant[i].nextID].sumRewardAgePerDA;
    uint rewardAfterD = participant[i].stake * deltaRewardAgePerDA;
    // Add i's previous rewards to new ones
    return participant[i].reward + rewardBeforeD + rewardAfterD;
```

4. Notes

1. The `Withdraw` and `CollectReward` functions can be merged into a single function for simplicity and efficiency.

Algorithm: Merged `Withdraw` and `CollectReward`

```
function Withdraw(block, i, amount) public:
    updateDepositAge(block, i);
    // Subtract amount from participant[i].reward first, then subtract remainder from participant[i].stake
    if amount > participant[i].reward then
        balance = participant[i].stake + participant[i].reward;
        if balance < amount then
            revert(); // Not enough balance
        participant[i].stake = balance - amount;
        T = T + participant[i].reward - amount;
        participant[i].reward = 0;
    else
        participant[i].reward -= amount;

function Balance(i) public:
    return participant[i].stake + Reward(i);
```

2. The current algorithm does not support compounding rewards, but there is potential for future development.

3. Participants who staked their tokens during a given distribution interval will earn rewards, even if they did not have an active stake at the time of the reward distribution. For example, if they withdrew their tokens before distribution, they will still receive their rewards after the distribution. This is an expected behavior because the described algorithm aims to provide fair reward distribution to all participants.

4. The algorithm is designed to be loop-free, ensuring efficient and scalable reward distribution.

5. Conclusion

This paper proposed a novel algorithm for efficient and decentralized reward distribution in smart contracts. The proposed algorithm overcomes several limitations of existing algorithms by handling variable reward rates based on the amount of reward tokens needed to be distributed while retaining O(1) time complexity.

In the context of DeFi protocols, efficient and fair reward distribution is crucial for their sustainability and growth. The proposed algorithm provides a promising solution to address the challenges of gas constraints, varying reward rates, and scaling to a large number of participants. It can contribute to the development of a more robust and efficient DeFi ecosystem.

References

[1] Vogelsteller, F. & Buterin, V. (2015). ERC: Token standard 20. Retrieved from `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`.

[2] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. Ethereum white paper. Retrieved from `https://ethereum.org/en/whitepaper/`.

[3] Batog, B., Boca, L., & Johnson, N. (2018) Scalable reward distribution on the Ethereum blockchain. *DappCon*, Berlin, Germany, Aug. 2018.

[4] Synthetix. (2021). StakingRewards Contract. Retrieved from `https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol`.

[5] SushiSwap. (2021). MasterChef Contract. Retrieved from `https://github.com/sushiswap/sushiswap/blob/master/protocols/masterchef/contracts/MasterChefV2.sol`.

[6] Merkle, R. C. (1988). A Digital Signature Based on a Conventional Encryption Function. *Advances in Cryptology — CRYPTO '87*. Lecture Notes in Computer Science. Vol. 293. pp. 369–378.

[7] Uniswap. (2020). MerkleDistributor contract. Retrieved from `https://github.com/Uniswap/merkle-distributor/blob/master/contracts/MerkleDistributor.sol`.