An $O(1)$ Token Distribution Algorithm for Smart Contracts

Ivan Menshchikov, Roman Vinogradov *

juglipaff@gmail.com

April 12, 2023

Abstract

This paper presents an algorithm for efficient decentralized distribution of tokens among pool participants at any time intervals, 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 compares demonstrates the 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 [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, which is not feasible for most projects due to gas constraints. Synthetix's StakingRewards [4] contract and Sushi's MasterChief [5] both automatically distribute rewards at a constant time intervals (e.g. every block), which does not work for yield farming projects with variable token rewards and time intervals between reward distributions.

In this paper, we propose a novel algorithm for decentralized reward distribution in smart contracts that overcomes these limitations while retaining $O(1)$ time complexity. Our algorithm can handle random time intervals with variable rewards for each token distribution and is particularly useful for yield farming projects that have long time intervals between token distributions. We also discuss the implementation of our algorithm as a smart contract and also compare our 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, we provide an overview of the related work on reward distribution in smart contracts.

*Special thanks to Anton Dementyev

2.1. Constant Reward Rate Algorithms

Several DeFi protocols use constant reward rate algorithms for reward distribution and distribute constant rewards at constant time intervals (e.g. every block). For example, Sushiswap's MasterChief [5] belongs to this category. Although constant reward rate algorithms are simple and easy to implement, they do not work well for yield farming projects with variable token rewards and time intervals.

2.2. Time-Based Algorithms

Another class of reward distribution algorithms is based on time intervals. These algorithms distribute varying 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 or random time intervals, the need for manual intervention to adjust the distribution frequency, and the potential for reward 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 because they are simple to implement. They calculate reward distribution information off-chain and use cryptographic proofs for verification. However, if the off-chain storage and calculation of such an algorithm is controlled by a single entity e.g. a centralized server, then this would be a centralized solution.

2.4. Share-Based Algorithms

Some token distribution algorithms only use participants' shares in their calculation of rewards. Such algorithm suffer from frontrunning issues, where malicious participants could join the pool just before reward distributions and receive the rewards they did not earn. This happens because such algorithms do not track the time when participants enter the pool and therefore do not work well with long distribution intervals. An example of such an algorithm is *Scalable Reward Distribution on the Ethereum Blockchain* (Batog, Boca & Johnson, 2018) [3].

2.5. Our Solution

While several algorithms have been developed to handle reward distribution in smart contracts, most of them suffer from significant limitations. Our 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

We will begin by outlining the key components of the algorithm:

- `Deposit(amount)` : This function enables participants to deposit their tokens into the pool, updating their stake and the total amount of tokens deposited.
- `Withdraw(amount)` : This function allows participants to withdraw tokens from their stake, updating their stake and the total amount of tokens remaining in the distribution pool.
- `CollectReward(amount)` : This function allows participants to collect their rewards from the pool, which is calculated based on their stake, the duration their tokens have been in the pool, and the reward distributions they participated in.

- `Distribute(reward)` : This function distributes rewards among participants. The contract utilizes a pull-based system to maintain constant time complexity. Participants must subsequently call `CollectReward` to receive their rewards.

To prevent the issue described in the section 2.4, the algorithm must also track the times at which participants made their deposits. Several challenges make calculating rewards a participant should receive a non-trivial task:

1. A participant can make an unlimited number of deposits and withdrawals at different blocks and with varying amounts.
2. There can be an unlimited number of reward distributions with varying reward amounts at different time intervals.

3.1. Participant's Reward After 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 a deposit or withdrawal occurred, where n is the block at which a reward of R is distributed. A participant would have a stake of s_1, s_2, ..., s_n at each such block depending on their deposits and withdrawals.

Let's assign variables to the following expressions:

$$depositAge = \sum_{k=1}^{n} (s_k(t_k - t_{k-1})) \tag{1}$$

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

A participant will receive a reward of:

$$r = \frac{depositAge}{totalDepositAge} R \tag{3}$$

We can accumulate $totalDepositAge$ on each deposit and withdrawal, making it possible to compute (2) in constant time on participant's collection of rewards.

Since the stake of a participant is constant over the interval in which no deposits or withdrawals were made by them, we can rewrite (1) as:

$$depositAge = \sum_{p=1}^{L} (s_{B(p)}(B(p) - B(p-1))), \tag{4}$$

Here, L represents the number of deposits and withdrawals made by a participant and $B(p)$ represents the block the participant made action p. We can also accumulate $depositAge$ on participants' deposits and withdrawals, allowing us to compute (4) in constant time. Therefore, we can compute r in constant time as well.

3.2. Participant's Reward After Multiple Distributions

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

$$reward = \sum_{m=d}^{D} (\frac{depositAge_m}{totalDepositAge_m} R_m) \tag{5}$$

After the first distribution d, the participant will receive a reward of (3). If $m > d$, their stake would remain constant:

$$depositAge_m = S\Delta t_m \tag{6}$$

Here, S represents the stake of the participant after their last action in the distribution interval d, and Δt_m is the length of distribution interval m in blocks.

$$reward = r_d + S \sum_{m=d+1}^{D} (\frac{\Delta t_m}{totalDepositAge_m} R_m) \tag{7}$$

We can rewrite (7) as:

$$reward = r_d + S(\sum_{m=1}^{D} (\frac{\Delta t_m}{totalDepositAge_m} R_m) - \sum_{m=1}^{d} (\frac{\Delta t_m}{totalDepositAge_m} R_m)) \tag{8}$$

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

$$A_d = \sum_{m=1}^{d} (\frac{\Delta t_m}{totalDepositAge_m} R_m) \tag{9}$$

So (8) becomes:

$$reward = r_d + S(A_D - A_d) \tag{10}$$

We can store the last distribution D on each new distribution and store the next distribution d for each participant on each of their actions making it possible to calculate $A_D - A_d$ in constant time.

We can also calculate r_d in constant time by storing $\frac{R_d}{totalDepositAge_d}$ for each distribution d and multiplying it by participant's $depositAge$ on their collection of rewards. Therefore, it is also possible to compute $reward$ in constant time.

3.3. Other Cases

Every other case can be modeled by adding multiple (10) expressions together and substituting d for the next distribution after a participant's last action. The algorithm remains constant time because we can accumulate participant's rewards on each of their actions.

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

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

function CollectReward(block, participant, amount) public:
    updateDepositAge(block, participant); // Update on each action
    if amount > participant.reward then
        revert(); // Not enough reward balance
    participant.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, participant) internal:
    if participant.nextID == ID + 1 then
        // If the distribution did not happen after participant.lastUpdate we accumulate participant's
           deposit age
        participant.DA += participant.stake * (block - participant.lastUpdate);
    else
        // If the distribution has happened after participant.lastUpdate we update participant's reward and
           start accumulating participant's deposit age from zero
        participant.reward = Reward(participant);
        participant.DA = participant.stake * (block - d[ID].block);
    participant.nextID = ID + 1;
    participant.lastUpdate = block;
    totalDA += (block - lastUpdate) * T; // Accumulate total deposit age
    lastUpdate = block;

function Reward(participant) public:
    if participant.nextID == ID + 1 then
        // If the distribution after participant's last deposit did not yet happen
        return participant.reward;
    // Add remaining deposit age and calculate reward between participant's last update and the distribution
       after
    uint DA = participant.DA + participant.stake * (d[participant.nextID].block - participant.lastUpdate);
    uint rewardBeforeD = DA * d[participant.nextID].rewardPerDA;
    // Calculate reward from the distributions that have happened after the last user deposit
    uint deltaRewardAgePerDA = d[ID].sumRewardAgePerDA - d[participant.nextID].sumRewardAgePerDA;
    uint rewardAfterD = participant.stake * deltaRewardAgePerDA;
    // Add participant's previous rewards to new ones
    return participant.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, participant, amount) public:
    updateDepositAge(block, i);
    // Subtract amount from participant.reward first, then subtract remainder from participant.stake
    if amount > participant.reward then
        balance = participant.stake + participant.reward;
        if balance < amount then
            revert(); // Not enough balance
        participant.stake = balance - amount;
        T = T + participant.reward - amount;
        participant.reward = 0;
    else
        participant.reward -= amount;
function Balance(participant) public:
    return participant.stake + Reward(participant);
```

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. Author Information

This algorithm was developed for the *Uno.Farm* [8] yield farming protocol. *Uno.Farm* [8] is an example of the successful implementation of the algorithm described in this paper and utilizes it to distribute rewards from third-party staking pools among participants.

6. Conclusion

In the context of DeFi protocols, efficient and fair reward distribution is crucial for their sustainability and growth. 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 rewards and time intervals between distributions while retaining $O(1)$ time complexity, which addresses the challenges of scaling to a large number of participants. This algorithm 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`.

[8] Uno.Farm. Yield farming aggregator protocol. `https://uno.farm/`.