WATCHPUG / Pendle v2 / Part 2

[M-1] Wrong implementation of `PendleVotingControllerUpg#vote()` will revert when there are pools that get more weights than before

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingController/PendleVotingControllerUpg.sol#L69-L91

```
69 function vote(address[] calldata pools, uint64[] calldata weights) external {
70     address user = msg.sender;
71
72     require(weights.length == pools.length, "invaid array length");
73     require(vePendle.balanceOf(user) > 0, "zero vependle balance");
74
75     UserData storage uData = userData[user];
76     LockedPosition memory userPosition = _getUserVePendlePosition(user);
77
78     for (uint256 i = 0; i < pools.length; ++i) {
79         if (_isPoolActive(pools[i])) applyPoolSlopeChanges(pools[i]);
80     }
81
82     for (uint256 i = 0; i < pools.length; ++i) {

84             _modifyVoteWeight(user, pools[i], userPosition, weights[i]);
85     }
86
87     for (uint256 i = 0; i < pools.length; ++i) {

89             _modifyVoteWeight(user, pools[i], userPosition, weights[i]);
90     }
91 }
```

At L82-85, the pools that get more weights than before will be updated first, which we believe is a wrong implementation; it seems the original intention is to update the pools that get fewer weights first.

As a result, the transaction will revert at L212-216 with the error: "exceeded max weight".

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingController/VotingControllerStorageUpg.sol#L181-L220

```
181 function _modifyVoteWeight(
182     address user,
183     address pool,
184         LockedPosition memory userPosition,
185         uint64 weight
186     ) internal returns (VeBalance memory newVote) {
187         UserData storage uData = userData[user];
188         PoolData storage pData = poolData[pool];
189 
190         VeBalance memory oldVote = uData.voteForPools[pool].vote;
191 
192         // REMOVE OLD VOTE
193         if (oldVote.bias != 0) {
194             if (_isPoolActive(pool) && _isVoteActive(oldVote)) {
195                 pData.totalVote = pData.totalVote.sub(oldVote);
196                 pData.slopeChanges[oldVote.getExpiry()] -= oldVote.slope;
197             }

199             delete uData.voteForPools[pool];
200         }
201 
202         // ADD NEW VOTE
203         if (weight != 0) {
204             require(_isPoolActive(pool), "pool not active");
205 
206             newVote = userPosition.convertToVeBalance(weight);
207 
208             pData.totalVote = pData.totalVote.add(newVote);
209             pData.slopeChanges[newVote.getExpiry()] += newVote.slope;
210 
211             uData.voteForPools[pool] = UserPoolData(weight, newVote);




217         }
218 
219         userPoolHistory[user][pool].push(newVote);
220     }
```

PoC

Given:

- Alice allocated weights as such: `[ pool1: 0.5, pool2: 0.5 ]`

When:

- Alice `vote()` with: `[ pool1: 0.4, pool2: 0.6 ]`

Then:

- `PendleVotingControllerUpg#vote()` the first iteration at L82-L85 will be skipped as `if (0.5 <= 0.4)` failed
- `PendleVotingControllerUpg#vote()` the second iteration at L82-L85 will call `_modifyVoteWeight(user1, pool2, userPosition, 0.6)` as `if (0.5 <= 0.6)` passed:
  - `_modifyVoteWeight()` L198, `uData.totalVotedWeight -= 0.5;` updated

`uData.totalVotedWeight` to `0.5`

- `_modifyVoteWeight()` L212, `uData.totalVotedWeight += 0.6;` updated `uData.totalVotedWeight` to `1.1`
- `_modifyVoteWeight()` L213-L216, reverted because of `require(1.1 <= 1.0, "exceeded max weight");`

Recommendation

- Moving the `uData.totalVotedWeight <= VeBalanceLib.USER_VOTE_MAX_WEIGHT` check in `VotingControllerStorageUpg#_modifyVoteWeight()` to the end of the `PendleVotingControllerUpg#vote()` ;
- Merging the two for-loop in `PendleVotingControllerUpg#vote()` into one:

```solidity
function vote(address[] calldata pools, uint64[] calldata weights) external {
        address user = msg.sender;

        require(weights.length == pools.length, "invaid array length");
        require(vePendle.balanceOf(user) > 0, "zero vependle balance");

        UserData storage uData = userData[user];
        LockedPosition memory userPosition = _getUserVePendlePosition(user);

        for (uint256 i = 0; i < pools.length; ++i) {
            if (_isPoolActive(pools[i])) applyPoolSlopeChanges(pools[i]);
        }




      }
```

```solidity
function _modifyVoteWeight(
        address user,
        address pool,
        LockedPosition memory userPosition,
        uint64 weight
    ) internal returns (VeBalance memory newVote) {
        UserData storage uData = userData[user];
        PoolData storage pData = poolData[pool];

        VeBalance memory oldVote = uData.voteForPools[pool].vote;

        // REMOVE OLD VOTE
        if (oldVote.bias != 0) {
            if (_isPoolActive(pool) && _isVoteActive(oldVote)) {
                pData.totalVote = pData.totalVote.sub(oldVote);
                pData.slopeChanges[oldVote.getExpiry()] -= oldVote.slope;
            }
            uData.totalVotedWeight -= uData.voteForPools[pool].weight;
            delete uData.voteForPools[pool];
        }

        // ADD NEW VOTE
        if (weight != 0) {
            require(_isPoolActive(pool), "pool not active");

            newVote = userPosition.convertToVeBalance(weight);

            pData.totalVote = pData.totalVote.add(newVote);
            pData.slopeChanges[newVote.getExpiry()] += newVote.slope;

            uData.voteForPools[pool] = UserPoolData(weight, newVote);
            uData.totalVotedWeight += weight;
        }

        userPoolHistory[user][pool].push(newVote);
    }
```

[L-2] Duplicate event emissions

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingEscrow/VotingEscrowPendleMainchain.sol#L213-L231

```
function _broadcastPosition(address user, uint256[] calldata chainIds) public payable {
        require(chainIds.length != 0, "empty chainIds");

        (VeBalance memory supply, uint256 wTime) = _applySlopeChange();

        bytes memory userData = (
            user == address(0) ? EMPTY_BYTES : abi.encode(user, positionData[user])
        );

        for (uint256 i = 0; i < chainIds.length; ++i) {
            require(sidechainContracts.contains(chainIds[i]), "not supported chain");
            _broadcast(chainIds[i], wTime, supply, userData);
```

```
        }

        emit BroadcastTotalSupply(supply, chainIds);
    }
```

`BroadcastUserPosition` will be emited `chainIds.length` times with all the `chainIds` each time.

Recommendation

```
function _broadcastPosition(address user, uint256[] calldata chainIds) public payable {
        require(chainIds.length != 0, "empty chainIds");

        (VeBalance memory supply, uint256 wTime) = _applySlopeChange();

        bytes memory userData = (
            user == address(0) ? EMPTY_BYTES : abi.encode(user, positionData[user])
        );

        for (uint256 i = 0; i < chainIds.length; ++i) {
            require(sidechainContracts.contains(chainIds[i]), "not supported chain");
            _broadcast(chainIds[i], wTime, supply, userData);
        }

```

```
        emit BroadcastTotalSupply(supply, chainIds);
    }
```

[I-3] Expired markets should be excluded from PENDLE rewards automatically

Expired markets should no longer receive any PENDLE rewards as the market is usually no longer needed by then.

Per the README.md:

> markets that are expired will be removed by governance

While this gives us more flexibility, we also believe it is prone to delay/mistake more than an automatic method.

Recommendation

Consider adding a check in `_receiveVotingResults` and only `_addRewardsToMarket` when the market is not expired. The `pendleAmounts` allocated to the expired markets can be sent to the treasury.

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/GaugeController/PendleGaugeControllerBaseUpg.sol#L93-L109

```
function _receiveVotingResults(
    uint128 wTime,
    address[] memory markets,
    uint256[] memory pendleAmounts
) internal {
    require(markets.length == pendleAmounts.length, "invalid markets length");

    if (epochRewardReceived[wTime]) return; // only accept the first message for the wTime
    epochRewardReceived[wTime] = true;

    for (uint256 i = 0; i < markets.length; ++i) {
        _addRewardsToMarket(markets[i], pendleAmounts[i].Uint128());
    }

    emit ReceiveVotingResults(wTime, markets, pendleAmounts);
}
```

If we want a more percise time to end the rewards for the soon-to-expire markets, `_addRewardsToMarket()` can be changed to reaplce `WEEK` with the length of time until it expires.

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/GaugeController/PendleGaugeControllerBaseUpg.sol#L117-L128

```
function _addRewardsToMarket(address market, uint128 pendleAmount) internal {
        MarketRewardData memory rwd = _getUpdatedMarketReward(market);
        uint128 leftover = (rwd.incentiveEndsAt - rwd.lastUpdated) * rwd.pendlePerSec;


        rewardData[market] = MarketRewardData({
            pendlePerSec: newSpeed,
            accumulatedPendle: rwd.accumulatedPendle,
            lastUpdated: uint128(block.timestamp),

        });
    }
```

[I-4] It's possible that the pool address can be the same on different networks, and we should avoid that

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingController/VotingControllerStorageUpg.sol#L136-L157

```
function _addPool(uint64 chainId, address pool) internal {
    require(chainPools[chainId].add(pool), "IE");
    require(allActivePools.add(pool), "IE");

    poolData[pool].chainId = chainId;
    poolData[pool].lastSlopeChangeAppliedAt = WeekMath.getCurrentWeekStart();
}

/**
```

```
* @dev expected behavior:
    - remove from allActivePool, chainPools
    - add to allRemovedPools
    - clear all params in poolData
    */
function _removePool(address pool) internal {
    uint64 chainId = poolData[pool].chainId;
    require(chainPools[chainId].remove(pool), "IE");
    require(allActivePools.remove(pool), "IE");
    require(allRemovedPools.add(pool), "IE");

    delete poolData[pool];
}
```

The address of the pool is used as the ID for the all the markets cross the networks, it assumes that the address is unique.

However, We find that it's possible for the address of a pool to be the same on another network, this can be troublesome, so we should avoid that.

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/Market/PendleMarketFactory.sol#L76-L95

```
function createNewMarket(
        address PT,
        int256 scalarRoot,
        int256 initialAnchor
    ) external returns (address market) {
        require(IPYieldContractFactory(yieldContractFactory).isPT(PT), "Invalid PT");
        require(markets[PT][scalarRoot][initialAnchor] == address(0), "market already created"

        // no need salt since market's existence has been checked before hand
        market = SSTORE2Deployer.create2(
            marketCreationCodePointer,
            bytes32(0),
            abi.encode(PT, scalarRoot, initialAnchor, vePendle, gaugeController)
        );

        markets[PT][scalarRoot][initialAnchor] = market;
        require(allMarkets.add(market), "IE market can't be added");

        emit CreateNewMarket(market, PT, scalarRoot, initialAnchor);
    }
```

When the `PendleMarketFactory` is deployed by the same deployer with the same nonce on a different network, AND when `createNewMarket` is called with the same params, then the pool address will be the same on different networks.

This is unlikely to happen in practice, but still possible.

Recommendation

Consider using the `chainId` as the salt in `createNewMarket()`:

```
function createNewMarket(
    address PT,
    int256 scalarRoot,
    int256 initialAnchor
) external returns (address market) {
    require(IPYieldContractFactory(yieldContractFactory).isPT(PT), "Invalid PT");
    require(markets[PT][scalarRoot][initialAnchor] == address(0), "market already created");

    // no need salt since market's existence has been checked before hand
    market = SSTORE2Deployer.create2(
        marketCreationCodePointer,

        abi.encode(PT, scalarRoot, initialAnchor, vePendle, gaugeController)
    );

    markets[PT][scalarRoot][initialAnchor] = market;
    require(allMarkets.add(market), "IE market can't be added");

    emit CreateNewMarket(market, PT, scalarRoot, initialAnchor);
}
```

[L-5] `PendleVotingControllerUpg.sol#_broadcastResults()` Precision loss due to `div` before `mul`

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingController/PendleVotingControllerUpg.sol#L245-L249

```
for (uint256 i = 0; i < length; ++i) {
    uint256 poolVotes = weekData[wTime].poolVotes[pools[i]];
    uint256 pendlePerSec = (totalPendlePerSec * poolVotes) / totalVotes;
    totalPendleAmounts[i] = pendlePerSec * WEEK;
}
```

Recommendation

Change to:

```
for (uint256 i = 0; i < length; ++i) {
    uint256 poolVotes = weekData[wTime].poolVotes[pools[i]];
    totalPendleAmounts[i] = (totalPendlePerSec * poolVotes) * WEEK / totalVotes;
}
```

[I-6] The external rewards should not be distributed according to vePENDLE shares based `activeBalance`

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/PendleGauge.sol#L66-L94

```
function _updateUserActiveBalancePrivate(address user) private {
    assert(user != address(0) && user != address(this));

    uint256 lpBalance = _stakedBalance(user);
    uint256 veBoostedLpBalance = _calcVeBoostedLpBalance(user, lpBalance);

    uint256 newActiveBalance = Math.min(veBoostedLpBalance, lpBalance);

    totalActiveSupply = totalActiveSupply - activeBalance[user] + newActiveBalance;
    activeBalance[user] = newActiveBalance;
}

function _calcVeBoostedLpBalance(address user, uint256 lpBalance)
    internal
    virtual
    returns (uint256)
{
    uint256 vePendleBalance = vePENDLE.balanceOf(user);
    uint256 vePendleSupply = vePENDLE.totalSupplyCurrent();
    // Inspired by Curve's Gauge
    uint256 veBoostedLpBalance = (lpBalance * TOKENLESS_PRODUCTION) / 100;
    if (vePendleSupply > 0) {
        veBoostedLpBalance +=
            (((_totalStaked() * vePendleBalance) / vePendleSupply) *
                (100 - TOKENLESS_PRODUCTION)) /
            100;
    }
    return veBoostedLpBalance;
}
```

Per the README.md:

> The Gauge/Market will receive rewardTokens from SCY as well as claiming the PENDLE token from gauge controller. All of the reward tokens (including PENDLE) will be distributed with boosting mechanism

While it's natural and reasonable for the PENDLE rewards to be distributed according to the `activeBalance`, we find it's quite unconventional and troublesome if this also applies to the distribution of SCY external rewards.

Firstly and philosophically, we would say that the external rewards is bonded to the SCY, therefore, they should not be redistributed according to any other metrics besides the lpBalance.

Furthermore and more practically, the redistribution mechanism will effectively make

the calculation of the profit-and-loss much more complicated for the users.

That's because by the same time they get PENDLE rewards, they may gain more external rewards or lose part of their intital external rewards based on the amount of veBlance, in comparsion to just hold thier SCY tokens or the external rewards are distributed based on lpBalance.

[G-7] `vePENDLE.sol` Combine two external calls into one can save gas

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/PendleGauge.sol#L83-L84

```
uint256 vePendleBalance = vePENDLE.balanceOf(user);
    uint256 vePendleSupply = vePENDLE.totalSupplyCurrent();
```

Recommendation

Consider adding a new function called `totalSupplyAndBlanaceCurrent` in `vePENDLE`:

```
function totalSupplyAndBlanaceCurrent(address user) external view returns (uint128, uint128) {
    (VeBalance memory supply, ) = _applySlopeChange();

    uint128 userBalance = balanceOf(user);
    return (userBalance, supply.getCurrentValue());
}
```

[G-8] Avoid unnecessary storage read can save gas

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingEscrow/VotingEscrowPendleMainchain.sol#L191-L210

```
function _applySlopeChange() internal returns (VeBalance memory, uint128) {
        VeBalance memory supply = _totalSupply;
        uint128 wTime = lastSlopeChangeAppliedAt;
        uint128 currentWeekStart = WeekMath.getCurrentWeekStart();

        if (wTime >= currentWeekStart) {
            return (supply, wTime);
198            }
199
200            while (wTime < currentWeekStart) {

201                wTime += WEEK;
202                supply = supply.sub(slopeChanges[wTime], wTime);
203                totalSupplyAt[wTime] = supply.getValueAt(wTime);
204            }
205
206            _totalSupply = supply;

208

210        }
```

At L209, use `wTime` instead of `lastSlopeChangeAppliedAt` can avoid unnecessary storage read and save some gas.

[I-9] End the `Vote` a few hours earlier before the next week starts can help avoid leftover

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/GaugeController/PendleGaugeControllerBaseUpg.sol#L117-L128

```
117  function _addRewardsToMarket(address market, uint128 pendleAmount) internal {
118      MarketRewardData memory rwd = _getUpdatedMarketReward(market);

120      uint128 newSpeed = (leftover + pendleAmount) / WEEK;
121
122      rewardData[market] = MarketRewardData({
123          pendlePerSec: newSpeed,
124          accumulatedPendle: rwd.accumulatedPendle,
125          lastUpdated: uint128(block.timestamp),
126          incentiveEndsAt: uint128(block.timestamp) + WEEK
127      });
128  }
```

In the current implementation, the voting for allocation to the pools in the next week/epoch only ends by the start of the next week/epoch.

After the voting ends and the new epoch/week starts, the voting results can be synced to the side chains through a crosschain messaging protocol.

However, this will take some time, depends on when the sync can be triggered and how long it takes for the cross-chain message to be processed. This creates a gap between the end of the last reward period and the start of a new reward period, during that gap period, there will be no rewards.

An alternative solution would be to end the vote earlier, say a few hours, to allow the voting results to be synced sooner, and avoid the gap.

the voting results to be synced sooner, and avoid the gap.

Recommendation

To apply this alternative solution, a decent amount of code changes would be required, and even with this early result synchronisation mechanism implemented, we still can't guarantee no gap.

Therefore, we believe it's not unnecessary to make any changes for this. And we leave this issue as it is for your reference only.

[Q-10] `VotingEscrowPendleMainchain.sol` How will the `totalSupplyAt` be used for reward accounting

When the user `increaseLockPosition()` a new point will be added to the $veTotalSupply(t)$.

However, `totalSupplyAt` will not be updated. Instead, it will only be set per week at `wTime`.

This aligns with the comment indicating that it's mapping of `[wTime] => totalSupply`:

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingEscrow/VotingEscrowPendleMainchain.sol#L25-L27

```
    mapping(uint128 => uint128) public totalSupplyAt;
```

The comment also says that this "later can be used for reward accounting", but we would like to learn more about how this can be or will be used for reward accounting?

[I-11] `VotingEscrowPendleMainchain#_broadcast()` should refund the unspent crosschain message fee

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingEscrow/VotingEscrowPendleMainchain.sol#L238-L245

```
function _broadcast(
        uint256 chainId,
        uint256 wTime,
        VeBalance memory supply
        veBalance memory supply,
        bytes memory userData
    ) internal {
        _sendMessage(chainId, abi.encode(wTime, supply, userData));
    }
```

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/CelerAbstracts/CelerSenderUpg.sol#L28-L33

```
function _sendMessage(uint256 chainId, bytes memory message) internal {
        assert(sidechainContracts.contains(chainId));
        address toAddr = sidechainContracts.get(chainId);
        uint256 fee = celerMessageBus.calcFee(message);
        celerMessageBus.sendMessage{ value: fee }(toAddr, chainId, message);
    }
```

When a crosschain message is needed, the caller is required to pay for the message with `msg.value` .

However, in the current implementation, the unspent part (`msg.value - fee`) will not be refunded to `msg.sender` .

This is not a problem with the current implementation of Celer's `MessageBusSender.sol#calcFee()` :

https://github.com/celer-network/sgn-v2-contracts/blob/d20dfaed94019c0404af0c86fce6ccb4c71b4b0d/contracts/message/messagebus/MessageBusSender.sol#L108-L127

```
/**
    * @notice Calculates the required fee for the message.
    * @param _message Arbitrary message bytes to be decoded by the destination app contract.
    @ @return The required fee.
    */
function calcFee(bytes calldata _message) public view returns (uint256) {
    return feeBase + _message.length * feePerByte;
}



function setFeePerByte(uint256 _fee) external onlyOwner {
    feePerByte = _fee;
    emit FeePerByteUpdated(feePerByte);
}

function setFeeBase(uint256 _fee) external onlyOwner {
    feeBase = _fee;
    emit FeeBaseUpdated(feeBase);
}
```

Because the result of `calcFee()` will not change for the same message unless the owner changed the `feeBase` or `feePerByte` .

We observed the recent transactions on Celer's `MessageBusSender.sol` contract and it seems they rarely change them.

It may not continue to be so if Celer changes the way they update the `feeBase` or `feePerByte` .

Plus, we may choose to user another cross chain messaging provider later, which may have a more dynamic messaging fee.

Therefore, we still recommend you to add the logic to refund the unspend fee to the caller.

[G-12] `PendleERC20.sol` storage-pack `_status` with `_totalSupply` increases runtime gas consumption due to masking

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/PendleERC20.sol#L20-L30

```
contract PendleERC20 is Context, IERC20, IERC20Metadata {
    uint8 private constant _NOT_ENTERED = 1;
    uint8 private constant _ENTERED = 2;

    mapping(address => uint256) private _balances;

    mapping(address => mapping(address => uint256)) private _allowances;

```

`_status` , `_totalSupply` are not typically read/written together.

Specifically, the `_status` used for reentrancy guard will be read and write more often.

Per the Solidity docs:

https://docs.soliditylang.org/en/v0.8.10/internals/layout_in_storage.html#:~:text=When%20using%20elements%20that%20are%20smaller%20than%2032%20bytes%2C%20your%20contract%E2%80%99s%20gas%20usage%20may%20be%20higher

> **When using elements that are smaller than 32 bytes, your contract's gas usage may be higher**. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, **the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size**.

> It might be beneficial to use reduced-size types if you are dealing with storage values because the compiler will pack multiple elements into one storage slot, and thus, combine multiple reads or writes into a single operation. If you are
>
> not reading or writing all the values in a slot at the same time, this can have the opposite effect, though: **When one value is written to a multi-value storage slot, the storage slot *has to be read first and then combined with the new value* such that other data in the same slot is not destroyed.**

Every contract has structs that pack multiple fields into slots by using < 256b types. This saves slots but increases runtime gas consumption due to masking of shared slot variables while reading/writing individual variables. The impact is more significant where the shared slot variables are not typically read/written together in functions which may allow the optimizer to combine their reads/writes in SLOADs/SSTOREs because that will not reduce SLOADs/SSTOREs used and instead add more bytecode/gas overhead for masking.

As a result, packing them into the same slot actually increases runtime gas consumption.

[I-13] Adding mainchain `gaugeController` into `sidechainContracts` is confusing and unnecessary

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/VotingController/PendleVotingControllerUpg.sol#L231-L263

```
function _broadcastResults(
    uint64 chainId,
    uint128 wTime,
    uint128 totalPendlePerSec
) internal {
    uint256 totalVotes = weekData[wTime].totalVotes;
    if (totalVotes == 0) return;

    uint256 length = chainPools[chainId].length();
    if (length == 0) return;

    address[] memory pools = chainPools[chainId].values();
    uint256[] memory totalPendleAmounts = new uint256[](length);

    for (uint256 i = 0; i < length; ++i) {
        uint256 poolVotes = weekData[wTime].poolVotes[pools[i]];
        uint256 pendlePerSec = (totalPendlePerSec * poolVotes) / totalVotes;
        totalPendleAmounts[i] = pendlePerSec * WEEK;
    }

    if (chainId == block.chainid) {
        address gaugeController = sidechainContracts.get(chainId);
        IPGaugeControllerMainchain(gaugeController).updateVotingResults(
            wTime,
            pools,
            totalPendleAmounts
        );
    } else {
        _sendMessage(chainId, abi.encode(wTime, pools, totalPendleAmounts));
    }

    emit BroadcastResults(chainId, wTime, totalPendlePerSec);
}
```

Per the README.md:

> On Ethereum, there is a `VotingController` contract to control the voting on incentives for the different markets on the different chains.

When `chainId == block.chainid`, it means that we are `updateVotingResults()` for the current network, the "mainchain".

The current implementation requires the mainchain's `gaugeController` address to be added to `sidechainContracts`, this may cause some misunderstandings as it's not extract how the name implies.

For example, `getAllSidechainContracts()` will also return mainchain's `gaugeController` address:

https://github.com/pendle-finance/pendle-core-internal-v2/blob/6b7ec5e22cc07617fd531dc71b1b2d2031aa68fe/contracts/core/LiquidityMining/CelerAbstracts/CelerSenderUpg.sol#L46-L58

```
function getAllSidechainContracts()
    public
    view
    returns (uint256[] memory chainIds, address[] memory addrs)
{
    uint256 length = sidechainContracts.length();
    chainIds = new uint256[](length);
    addrs = new address[](length);

    for (uint256 i = 0; i < length; ++i) {
        (chainIds[i], addrs[i]) = sidechainContracts.at(i);
    }
}
```

Recommendation

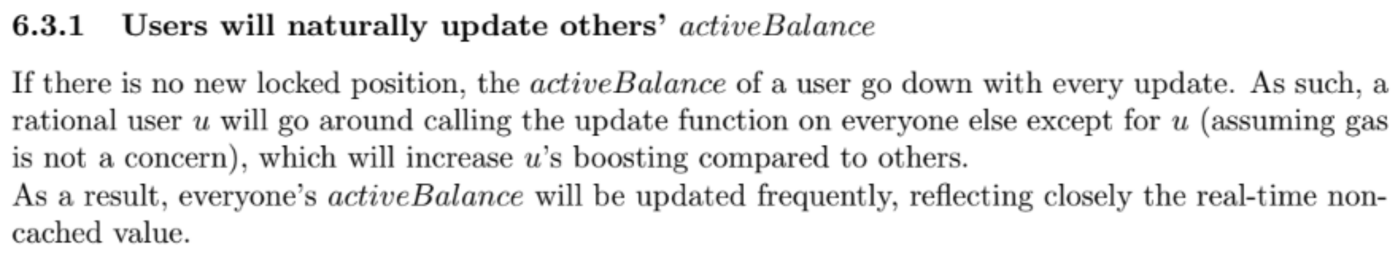
Consider adding a immutable variable `gaugeControllerMainchain` to store the mainchain's `gaugeController` address and only include sidechain contracts in `sidechainContracts`.

[I-14] Users will not naturally update others'

activeBalance

Work in progress

vePENDLE Whitepaper assumes that `Users will naturally update others' activeBalance` :



6.3.1 Users will naturally update others' *activeBalance*

If there is no new locked position, the *activeBalance* of a user go down with every update. As such, a rational user u will go around calling the update function on everyone else except for u (assuming gas is not a concern), which will increase u's boosting compared to others.
As a result, everyone's *activeBalance* will be updated frequently, reflecting closely the real-time non-cached value.

However, we find that it's more complicated than that and we are working on the details about this.