Morpho Optimizer Yellow Paper

Version 1.1

Mathis Gontier Delaunay[*]
Morpho Labs

Quentin Garchery
Morpho Labs

Paul Frambot
Morpho Labs

Merlin Égalité
Morpho Labs

Julien Thomas
Morpho Labs

Katia Babbar[†]
University of Oxford

September 5, 2022

Abstract

The Morpho protocol is built on top of existing pool-based lending protocols, improving rates for both lenders and borrowers while preserving the same liquidity and liquidation parameters. This paper aims to provide a detailed description of the Morpho protocol, notably the innovative mechanisms that make it work, along with invariants and theorems about it.

[*] Head of research and corresponding author, please direct any inquiry to mathis@morpho.xyz

[†] Visiting Academic

Contents

Acknowledgments

This work would not have been possible without the involvement of the rest of the Morpho Labs team and the advisors of the project, including but not limited to Vincent Danos, Adrien Husson, Jean Krivine from the Mangrove team, and a16z research.

Introduction

This paper, referenced as the yellow paper, assumes that the reader is familiar with the general idea behind the Morpho protocol. To strengthen their intuition about the Morpho protocol, the reader may first go through the white paper [DFB+21].

Compound and Aave ([GWPK20, PWXL20]) have established themselves as the current standard for decentralized lending protocols over the past years. These protocols, operating autonomously on-chain, implement a liquidity pool that offers strong liquidity guarantees and floating rates. Different protocols have been designed to improve on PLFs, protocols for loanable funds, in different aspects. One of these aspects is going from floating to fixed rates, and many protocols are tackling this issue. For example, APWine [APW22], Sense [Sen22], and 88mph [88m22] allow the user to trade their yield. Yield Protocol [RN20] and Notional [Not22] define a similar concept to collateralized zero-coupon bonds.

The Morpho protocol, for its part, leverages the composability and liquidity of existing PLFs, to create efficient and liquid peer-to-peer markets of supply and borrow positions with near-zero spread. The claim is that Morpho strictly improves pool-based liquidity protocols from an individual user perspective. In particular, Morpho has the same liquidation parameters and benefits from the same liquidity of the underlying protocols.

The first goal of this paper is to provide a comprehensive description of the Morpho protocol and its unique mechanisms. Then, to demonstrate the claims, this paper will provide mathematical properties and proofs about the logic underpinning the protocol. The aim is to bring transparency while educating and providing trust for the community to the key variables and invariant assumptions it utilizes.

The paper structure is as follows. In section 1, we detail the logical framework we place ourselves into. Our formalization is based on a transition system with transition functions written in pseudo-code. Section 2 describes the on-chain environment that will be useful for Morpho, including the involved contracts' functions. Section 3 details the structure and operation of Morpho, giving us a basis to analyze and prove some of its relevant invariants. Section 4 looks at the matching engine at the heart of the protocol, allowing matching and unmatching users when needed. The matching engine is voluntarily left abstract. The intent is to allow for different implementations. Section 5 focuses on the interest rates mechanism and their associated accounting. Finally, section 6 presents the delta mechanism allowing the protocol to scale despite the limited computation power available in a blockchain setting.

An emphasis is put on the correctness of the protocol, and different aspects of it are detailed along the way: balances integrity, covering borrow, supply, and aggregated peer-to-peer integrity, Morpho's positions integrity on the pool, as well as the liquidity of the positions.

1 Logical framework

In this first section, we aim to give a high-level description of the setting to reason about the Morpho protocol and its environment. We first detail the basic functionalities of the blockchain that we will build upon, which allows us to then describe our formalization of the blockchain in terms of a transition system. The formal description is close to the code, with sufficient details for describing invariants and proofs.

1.1 Definitions

We place ourselves in the blockchain setting where we have access to a set of **addresses** A. An address[1] $a \in A$ refers either to a **contract** or a user. For the sake of simplicity, the term ‘user‘ and the address associated with their account on the blockchain will be used interchangeably throughout this yellow paper.

Contracts are defined by their state variables and functions, similar to attributes and methods in object-oriented programming. From its deployment, a contract stores its variables, which are modified later by the function calls made by the users. Types take one of the following form:

- Primitive types T: booleans (B), integers ($\mathbb{N}$), real numbers ($\mathbb{R}$), non-negative real numbers ($\mathbb{R}_+$) and addresses (A). Notice that instead of using fixed-point arithmetic, we introduce real numbers to model the variables in question.

- Products $T_1 \times T_2$ of two primitive types, T_1 and T_2: pairs of a value in T_1 and a value in T_2. We will denote products by separating the values by commas. We identify products up to associativity such that we can omit parenthesis to disambiguate the order of a product: $T_1 \times (T_2 \times T_3)$ is the same as $(T_1 \times T_2) \times T_3$ and will be written as $T_1 \times T_2 \times T_3$.

- Mappings $T_1 \to T_2$ from a primitive type T_1 to a type T_2: vectors of the space of sequences indexed by T_1 with values in T_2. For a mapping m and a key i, $m(i)$ describes the value associated to i in m. If v is in T_2, then $m[i \mapsto v]$ refers to the mapping m where the value associated to i is v instead of $m(i)$.

[1] in EVM instances: 160 bits identifiers

Each contract has its own set of variables, which we will call the contract's **storage**, accessible through its address. We place ourselves in a simpler setting where the storage is the association of variables to their values for all the variables of all the contracts. Storage variables are thus directly accessible through their name in the storage. In this way, the mapping notation will also be used for storage, so for κ as the storage, the notation $\kappa(i)$ refers to the value of the variable i stored in κ. To keep the notation concise, we often omit the storage and write i for the value of the variable i.

Functions may read and modify storage variables, and we model this behavior by a transition system.

1.2 Transition system

The blockchain can be seen as a transition system where the state consists of the storage and the time, and the transition function is determined from the functions of the contracts. We use the following notations when a function f is called:

- $d \in D_f$ is the function input given by the caller, D_f is the domain of f
- $\sigma_i \in \Sigma$ is the current state, Σ is the set of states
- $r \in R_f$ is the output of the function, R_f is the image of f
- $\sigma_r \in \Sigma$ is the new (changed) state.

$$f : D_f \to R_f$$

The transition function Γ is defined in the following way. For each function call, we add a transition from σ_i to σ_r labeled by (d, r). All the transitions added for f can be described as a transition function Γ_f, where $\Gamma_f(d, \sigma_i) = (r, \sigma_r)$.

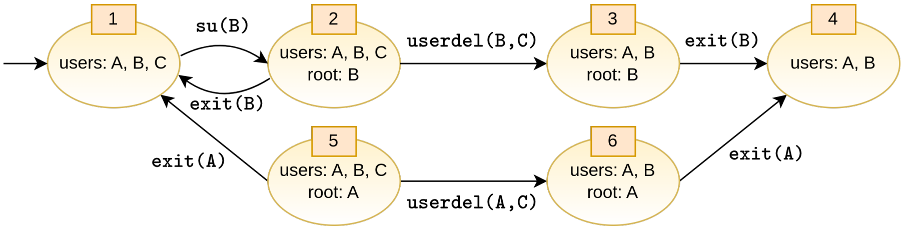
$$\Gamma_f : D_f \times \Sigma \to R_f \times \Sigma$$

Some function calls are only possible in some contexts for a particular configuration of the current state. In such a case the transition is not added. Moreover, some functions are defined by calling other functions. When the call of the callee function does not correspond to a transition, the call of the caller function also does not correspond to a new transition. This behavior mimics error handling in programming languages.

Example 1.2.1. *Let's give the example of a contract with user accounts and authorizations that is inspired from Unix systems. To be concise, we omit the particular implementation, and suppose that:*

- *In the initial state there are 3 normal accounts, account A, B and C. The account B is the owner of the contract.*
- *There is a function* `su` *that takes an account as an input parameter and promotes it to root. This function is only defined for the owner.*
- *There is a function* `exit` *that takes a root account as an input parameter and demotes it to a normal account.*
- *There is a function* `userdel` *that takes two accounts as input parameters. This function is only defined when the first account is root, and deletes the second account when this is the case. To limit the number of accessible states, we only define this function when the deleted account is the account C.*

The following figure represents this transition system. We label each transition with its corresponding function call for clarity and omit the returned value (as it is unused).



Notice that the user A is not able to become root, this is represented by the absence of a transition from state 1 *to state* 5*. This means that if the user A tries to call a function that batches* `su`*,* `userdel` *and* `exit` *(equivalent to the command* `sudo userdel C`*), this call would fail.*

Function calls do not change the time parameter, and we add one transition function that changes the time:

$$\Gamma_{tick} : \mathbb{1} \times \Sigma \to \mathbb{1} \times \Sigma$$

with $\Gamma_{tick}(0, (t, \kappa)) = (0, (t + 1, \kappa))$. We used the notation $\mathbb{1}$ for the singleton set containing 0. Thus, for a given path, multiple states can have the same time parameter. If we go back to the blockchain setting, this is useful to model the evolution of the storage inside one particular block. To keep it simple, we will write the time parameter as a subscript when there is no ambiguity about which storage it refers to or omit the time entirely when it is irrelevant. We will also allow ourselves to use the notation of the storage directly to the state. Notably, for $\sigma = (t, \kappa)$, the notation $\sigma(i)$ refers to $\kappa(i)$.

Example 1.2.2. *Consider a counter contract, whose storage is made of a single integer $i \in \mathbb{N}$, and which has a single function* `incrementBy`*, that increments a variable i by an input parameter $n \in \mathbb{N}$ and returns the new value. Here we have $D_{\texttt{incrementBy}} = \mathbb{N}$, $R_{\texttt{incrementBy}} = \mathbb{N}$, and:*

$$\Gamma_{\texttt{incrementBy}}(n, \sigma) = (\sigma(i) + n, \sigma[i \mapsto \sigma(i) + n])$$

In the following, we will only consider the transition functions Γ_f associated to a computable function f. Such functions can be described more simply by pseudo-code, where σ_i and σ_r store respectively the values of all the variables before and after the function call, and d and r are the input and the output of the function. When defining these functions, we will specify their domain and codomain (D_f and R_f) and assume that the function has access to the state.

Example 1.2.3. *The previous example could be more easily described by the following pseudo-code:*

```
function incrementBy(n):
    i += n
    return i
```

Example 1.2.4. *Define view functions*[2] *as functions that can read the state but cannot modify it. A function f is a view function if and only if for each argument d and storage σ, the second element of the pair $\Gamma_f(d, \sigma)$ is σ.*

[2] Inspired by Solidity's state mutability, as discussed in the Solidity documentation [sol22].

2 On-chain environment

The Morpho protocol uses two types of contracts: tokens[3] and PLFs, which are pool-based liquidity protocols. They constitutes the on-chain environment of Morpho, and we discuss the inner workings of such contracts in this section.

2.1 Tokens

Tokens are contracts that represent a transferrable store of value. We call Θ the set of tokens.

2.1.1 Storage

A token contract stores balances, which represent the amount held by each user.

Balances: The variable m, called the balances, is a mapping from addresses to real numbers.

$$m : A \to \mathbb{R}_+$$

2.1.2 Functions

A token contract can transfer funds from one address to another.

Transfer: Transfer `amount` $\in \mathbb{R}_+$ of the token balance of an address `from` $\in A$, to another address `to` $\in A$, where `amount` $\leq m($`from`$)$.

$$\texttt{transfer} : A \times A \times \mathbb{R}_+ \to \mathbb{1}$$

A simple implementation of the transfer function could look like the following:

```
transfer(from, to, amount):
    if amount > $m$(from):
        raise error
    $m$(from) -= amount
    $m$(to) += amount
```

[3] For EVM instances, ERC20 tokens [VB15].

Remark that the transfer is only possible when the sender has sufficient funds. In this case, it does not correspond to a new transition in the transition system.

2.2 Protocol for loanable funds

Let Ψ be the set of protocols for loanable funds[4]. For each contract $\psi \in \Psi$, we note Θ_ψ the subset of Θ of tokens available for lending and borrowing in ψ. A protocol $\psi \in \Psi$ should make available, for each token $\theta \in \Theta_\psi$, user entry points with functions to lend, borrow, withdraw lent funds, repay borrowed funds, and liquidate a position. In the following, we set the protocol ψ, which allows us to omit it in the notation. To be concise, we use the subscript notation to denote a function taking a token as an argument.

2.2.1 Storage

The storage of a protocol for loanable funds should store the balances of the borrowers and of the suppliers. It should also store the collateral factors of assets, which define the maximum debt a user can maintain. Liquidation of the position can happen if this limit is reached. See subsection 2.2.3 for more details about the `liquidate` function.

Collateral factor: For a given asset, the collateral factor[5] represents the maximum share of the value of the collateral that can be borrowed with this as collateral. The collateral factor values are fixed between 0 and 1, with 0 for assets that are not accepted as collateral[6].

$$F : \Theta_\psi \to [0, 1]$$

Example 2.2.1. *Alice deposits 1000* `DAI` *(worth 1000\$) on Compound, whose collateral factor is 80%. Alice can now borrow up to 800\$ worth of assets.*

[4] Aave and Compound belong to Ψ.

[5] Name given by Compound, Aave uses liquidation threshold (LT)

[6] For example, USDT at present on Aave and Compound

Rates: Supply and borrow rates, r^S and r^B, respectively, represent the growing speed of the balances per unit of time[7]:

$$r^S : \Theta_\psi \to \mathbb{R}_+$$
$$r^B : \Theta_\psi \to \mathbb{R}_+$$

Example 2.2.2. *If Alice deposits 1000* `DAI`*, and* $r^S_{\texttt{DAI}} = 0.01$*, after 10 units of time (second or block depending on the protocol), the accrued interest would be* $1000 \times (1 + r^S_{\texttt{DAI}})^{10} = 1104.62$.

Balances: The balances are mappings reflecting the users' balances, in a unit that grows in underlying value, at a speed that depends either on the supply rate r^S for the supply balance s or on the borrow rate r^B for the borrow balance b.

$$s : A \to \Theta_\psi \to \mathbb{R}_+$$
$$b : A \to \Theta_\psi \to \mathbb{R}_+$$

Indexes: The supply and borrow indexes, λ^S and λ^B, respectively, are the exchange rates between stored balances in the scaled unit and underlying tokens. According to the interest rate model, they constantly grow with time at a speed that depends on r^S and r^B, respectively, to accrue the interests of suppliers and borrowers. The amount of underlying tokens corresponding to an amount in the scaled unit is obtained by multiplying this last one by the corresponding index.

$$\lambda^S : \Theta_\psi \to \mathbb{R}_+$$
$$\lambda^B : \Theta_\psi \to \mathbb{R}_+$$

Having indexes to track interest rate accrual is particularly suitable in a blockchain environment, where computations are limited. It removes the need for an active accounting of interest accrual for every account or a history-based accounting required in a variable rate setting. By updating the indexes before each time that the rate changes, all users have their interests accrued at once.

[7] Second for Aave, and block for Compound

Example 2.2.3. *We expand on the previous example. Alice deposits* 1000 *DAI and, initially, the index* $\lambda^S_{\mathtt{DAI}}$ *is equal to* 2*, so her supply balance* $s_{\mathtt{DAI}}(Alice)$ *is* $1000\ /\ 2 = 500$.
After 10 units of time, the index $\lambda^S_{\mathtt{DAI}}$ *becomes* $2 \cdot (1 + r^S_{\mathtt{DAI}})^{10} = 2.20924$. *So her underlying balance is now:*

$$s_{\mathtt{DAI}}(Alice) \cdot \lambda^S_{\mathtt{DAI}} = 500\ \cdot\ 2.20924 = 1104.62\ \mathtt{DAI}$$

We first describe how the storage is managed internally, with the so-called utils functions, before describing the main entry points made accessible to the users.

2.2.2 Utils Functions

Update rates: The function update_r updates the rates for one market depending on the utilization rate[8].

$$\mathtt{update\_}r : \Theta_\psi \to \mathbb{1}$$

Update indexes: The function update_λ updates exchanges rates between users' balances and underlying tokens, depending on the rates since the last update.

$$\mathtt{update\_}\lambda : \Theta_\psi \to \mathbb{1}$$

Exchange rates are constantly updated before rates change.

Price oracle: The oracle function p returns the current price of an asset. It corresponds to an estimated price in a fixed normalizing asset or currency.

$$p : \Theta_\psi \to \mathbb{R}_+$$

The price oracle gives a way to compute the value of a particular asset. Oracles are provided by a source that is external to the lending pool and can be updated at regular intervals or according to the relative change in value. Decentralization, trust, and reliability are essential properties of the oracles, and the integrity of protocols such as the PLFs depends on them[9]. Popular oracles include the ChainLink oracle [SEN17] and the Uniswap oracle [Tea].

[8] See Compound's documentation [Com22], and Aave's documentation [Aav22] for the specific calculations

[9] Price manipulations should be considered a source of risk for the protocols that depend on oracles.

Borrowing capacity: The function `borrowing_capacity` computes and returns the borrowing capacity (in value) of a given user, which is the amount that can be borrowed in addition to the user's current debt.

$$\texttt{borrowing\_capacity} : A \to \mathbb{R}$$

This function first updates all the indexes it needs to access by calling the function $\texttt{update\_}\lambda$ on the respective markets. It then computes its result according to the following formula:

$$\texttt{borrowing\_capacity}(a) = \sum_{\theta \in \Theta_\psi} s_\theta(a) \cdot \lambda_\theta^S \cdot p_\theta \cdot F_\theta - \sum_{\theta \in \Theta_\psi} b_\theta(a) \cdot \lambda_\theta^B \cdot p_\theta$$

2.2.3 Main Entry Points

In the following functions, the address $\overline{\psi}$ refers to the address of the underlying protocol for loanable funds ψ.

Supply [10]**:** The `supply` function is used to deposit an amount $\texttt{amount} \in \mathbb{R}_+$ of tokens $\theta \in \Theta_\psi$ into the protocol, which begins accumulating interests at the supply rate.

$$\texttt{supply} : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

A standard implementation is given by:

```
supply$_\theta$(user, amount):
    update_$\lambda_\theta$()
    transfer$_\theta$(user, $\overline{\psi}$, amount)
    $s_\theta$(user) += amount / $\lambda_\theta^S$
    update_$r_\theta$()
```

Remark that because the `transfer` function can raise an error, some calls to `supply` can also raise an error. It happens when the user has insufficient funds, and we do not add a new transition in the transition system in this case.

[10] Compound calls it mint in reference to the fact that cTokens are minted during this operation, and Aave calls it deposit.

Borrow: The `borrow` function is used to borrow an amount `amount` $\in \mathbb{R}_+$ of tokens $\theta \in \Theta_\psi$ from the protocol.

$$\texttt{borrow} : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

An implementation enforcing the borrowing limit could look like the following. Note that we should also ensure that the amount is less than the available supply.

```
borrow$_\theta$(user, amount):
    update_$\lambda_\theta$()
    if amount * $p_\theta$ $\leq$ borrowing_capacity(user):
        transfer$_\theta$($\overline{\psi}$, user, amount)
        $b_\theta$(user) += amount / $\lambda_\theta^B$
        update_$r_\theta$()
```

Withdraw [11]**:** The `withdraw` function is used to withdraw previously supplied assets to the protocol [12].

$$\texttt{withdraw} : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

The following implementation makes sure that the amount withdrawn is small enough such that the borrowing capacity is still greater than 0.

```
withdraw$_\theta$(user, amount):
    update_$\lambda_\theta$()
    if amount * $p_\theta$ * $F_\theta$ $\leq$ borrowing_capacity(user):
        transfer$_\theta$($\overline{\psi}$, user, amount)
        $s_\theta$(user) -= amount / $\lambda_\theta^S$
        update_$r_\theta$()
```

Repay: The `repay` function is used to repay some accumulated debt to the protocol.

$$\texttt{repay} : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

The following implementation enforces that users do not repay more than their current debt (what has been borrowed plus the accrued interest).

[11] Compound calls it redeem.

[12] Note that this can be seen as converting c/aTokens into underlying tokens.

```
repay$_\theta$(user, amount):
    update_$\lambda_\theta$()
    if amount $\leq b_\theta$(user) * $\lambda_\theta^B$:
        transfer$_\theta$(user, $\overline{\psi}$, amount)
        $b_\theta$(user) -= amount / $\lambda_\theta^B$
        update_$r_\theta$()
```

Liquidate: The function `liquidate` liquidates a user's undercollateralized position. A liquidator repays a part of the user's debt and receives a part of the corresponding collateral backing the debt as an incentive. This process aims to deleverage the position to a safe debt ratio.

$$\texttt{liquidate} : \Theta_\psi \times \Theta_\psi \times A \times A \times \mathbb{R}_+ \to \mathbb{1}$$

A call to `liquidate(`θ_b`, `θ_c`, user, liquidator, toRepay)` requires the position of `user` to be undercollateralized. The `toRepay` parameter is the amount of debt that the `liquidator` repays on behalf of the `user`. This amount should not represent more than a fraction of the borrowed tokens θ_b. The close factor[13] variable `closeFactor` is a value between 0 and 1 representing this fraction.

The incentive for the liquidator is to be able to seize an amount `toSeize` of the collateral tokens θ_c of the `user` that is greater (in value) than the debt repaid. The ratio of the seized collateral (in value) over the repaid debt (in value) is defined as the `liquidationIncentive`[14] parameter, a parameter greater than 1.

Here is a possible implementation of the `liquidate` function:

```
liquidate($\theta_b$, $\theta_c$, user, liquidator, toRepay):
    update_$\lambda_{\theta_b}$()
    update_$\lambda_{\theta_c}$()
    borrows = $b_{\theta_b}$(user) * $\lambda_{\theta_b}^B$
    if borrowing_capacity(user) < 0 and
           toRepay $\leq$ borrows * closeFactor:
```

[13] Name used by Compound and Aave, even though Aave has two values for it, depending on how "dangerous" the position is considered. See Compound's documentation [Com22] and Aave's documentation [Aav22].

[14] Aave calls it the `liquidationBonus`.

```
toSeize = (toRepay * $p_{\theta_b}$ / $p_{\theta_c}$) * liquidationIncentive
repayOnBehalfOf$_{\theta_b}$(user, liquidator, toRepay)
withdrawOnBehalfOf$_{\theta_c}$(user, liquidator, toSeize)
```

where `repayOnBehalfOf` and `withdrawOnBehalfOf` are the functions `repay` and `withdraw` but with the `liquidator` acting as the `user` for the transfer. Concretely, those functions take an additional parameter `liquidator` used to replace the `user` in the underlying transfer call of those functions. For instance, the `withdrawOnBehalfOf` function changes the line

```
transfer$_\theta$($\overline{\psi}$, user, amount)
```

into the line

```
transfer$_\theta$($\overline{\psi}$, liquidator, amount)
```

The purpose of the `closeFactor` is to ensure that an undercollateralized position does not get liquidated entirely. A liquidator has to use multiple calls with smaller amounts, potentially bringing the position gradually closer to a safe debt ratio.

We say that a position is insolvent when the value of the position's debt is greater than the value of the position's collateral. In that case, the borrower can leave with the borrowed assets as there is no incentive to repay. Therefore, insolvent positions are a liability for the protocol and should be avoided. Liquidations help to solve this issue by targeting positions with a negative borrowing capacity, trying to move them away from insolvency. A position is implied to be safe when its borrowing capacity is positive. There is a gap between safe and insolvent positions, which can be used as a metric to gauge how dangerous a position is.

Suppose a position is at the limit of insolvency and liquidation happens. Because of the `liquidationIncentive` factor, the liquidation removes more collateral than borrowed assets. The position is now insolvent. This example shows that liquidation does not always help put the position back in the safe zone, and we want to know when it is the case. To that end, let us first define the total value of the borrowed assets B by:

$$B = \sum_{\theta \in \Theta_\psi} b_\theta(user) \cdot \lambda_\theta^B \cdot p_\theta$$

and the total value of the supplied assets S by:

$$S = \sum_{\theta \in \Theta_\psi} s_\theta(user) \cdot \lambda_\theta^S \cdot p_\theta$$

Lemma 2.2.1. *The ratio $\frac{B}{S}$ decreases after a liquidation if and only if this ratio was under $\frac{1}{\texttt{liquidationIncentive}}$ before the liquidation.*

Proof. After liquidation, the quantity $\frac{B}{S}$ decreases when:

$$\begin{aligned}\frac{B}{S} &> \frac{\sum\limits_{\substack{\theta\in\Theta_\psi\\ \theta\neq\theta_b}} b_\theta(user)\cdot\lambda_\theta^B\cdot p_\theta \quad + \quad (b_{\theta_b}(user)\cdot\lambda_{\theta_b}^B - \texttt{toRepay})\cdot p_{\theta_b}}{\sum\limits_{\substack{\theta\in\Theta_\psi\\ \theta\neq\theta_c}} s_\theta(user)\cdot\lambda_\theta^S\cdot p_\theta \quad + \quad (s_{\theta_c}(user)\cdot\lambda_{\theta_c}^S - \texttt{toSeize})\cdot p_{\theta_c}}\\ &> \frac{B - \texttt{toRepay}\cdot p_{\theta_b}}{S - \texttt{toRepay}\cdot p_{\theta_b}\cdot\texttt{liquidationIncentive}}\end{aligned}$$

which is equivalent to

$$-B\cdot\texttt{toRepay}\cdot p_{\theta_b}\cdot\texttt{liquidationIncentive} > -S\cdot\texttt{toRepay}\cdot p_{\theta_b}$$

and to

$$\frac{B}{S} < \frac{1}{\texttt{liquidationIncentive}}$$

■

In the following, we will write $\mathcal{T}$ for $\frac{1}{\texttt{liquidationIncentive}}$. Suppose a user is liquidatable with a ratio $\frac{B}{S}$ that is under $\mathcal{T}$. Then, we know that after one liquidation, the ratio has decreased. Either the user is back in the safe zone, or its position is still liquidatable with a ratio that is still under $\mathcal{T}$. Consider now multiple liquidations happening one after the other on the user's position. Since there is an upper bound on the ratio $\frac{B}{S}$, we know the user will run out of borrowed assets before running out of collateral. So, if it is impossible to liquidate anymore, then either there are no more borrowed assets to repay, or the position is not liquidatable anymore. Either way, the result is a solvent position.

Suppose now that a user is liquidatable with a ratio over $\mathcal{T}$, and consider multiple liquidations happening on its position. The ratio will keep growing. Hence the user's position will stay liquidatable, and the liquidations will stop only when the user lacks collateral assets. The result is an insolvent position.

We conclude that it is preferable if liquidations happen before the threshold $\mathcal{T}$. This threshold is itself before insolvency ($\frac{B}{S} = 1$).

3 General description of Morpho

In this section, we describe the main functionalities of the Morpho protocol. We start by detailing the internals of Morpho: first, the storage variables of the contract, followed by its internal functions. This allows us to describe then the external functions in more detail. Those functions are the ones with which the users interact. We conclude this section by proving the integrity of Morpho on the underlying pool.

Given one protocol for loanable funds ψ, the corresponding Morpho protocol is noted M_ψ, and we refer to its contract address as $\overline{M_\psi}$. We denote by Θ_{M_ψ} the subset of Θ_ψ of tokens available on Morpho for the protocol ψ. For the sake of this paper, the particular PLF on top of which Morpho is built does not matter. For this reason, in the following, we set the protocol to be a generic ψ, which allows us to omit it in the notation. To remain concise, we use the subscript notation to denote a function taking a token as an argument or omit it entirely when it can be inferred from context.

3.1 Storage

In order to follow debt and supply increase over time, user balances are stored in *scaled units*, whose underlying value grows over time. Pool credit lines evolve at pool rates, and Morpho reuses the index mechanism of the underlying pool to track them. The protocol also has its own peer-to-peer scaled unit and associated index.

Supplier balances: Suppliers have two balances in Morpho. The pool supply balance s^{Pool} is in a unit that grows in underlying token value at r^S speed and represents the supply that is deposited on the pool. The matched or peer-to-peer supply balance s^{P2P} grows at r^{P2P} speed, and represents the matched peer-to-peer supply.

$$
\begin{aligned}
s^{Pool} &: \Theta_{M_\psi} \to A \to \mathbb{R}_+ \\
s^{P2P} &: \Theta_{M_\psi} \to A \to \mathbb{R}_+
\end{aligned}
$$

The balances separation is the mechanism that makes Morpho positions not fungible. The total supply balance of a user is the sum of the two balances brought back to the underlying unit. Let us use the following notations for the total supply balance of the user u:

$$\omega_\theta^S(u) = s_\theta^{Pool}(u) \cdot \lambda_\theta^S + s_\theta^{P2P}(u) \cdot \lambda_\theta^{P2P} \tag{1}$$

Borrower balances: The same mechanism is used for borrower balances. The pool borrow balance b^{Pool} represents the on-pool debt and grows at r^B speed, while the matched or peer-to-peer borrow balance b^{P2P} grows at r^{P2P} speed.

$$b^{Pool} : \Theta_{M_\psi} \to A \to \mathbb{R}_+$$
$$b^{P2P} : \Theta_{M_\psi} \to A \to \mathbb{R}_+$$

The total borrow balance of a user is the sum of the two balances brought back to the underlying unit. Let us use the following notations for the total borrow balance of the user u:

$$\omega_\theta^B(u) = b_\theta^{Pool}(u) \cdot \lambda_\theta^B + b_\theta^{P2P}(u) \cdot \lambda_\theta^{P2P} \tag{2}$$

Index: The peer-to-peer index λ^{P2P} is the exchange rate between stored peer-to-peer balances in the peer-to-peer scaled unit and underlying tokens. It constantly grows with time, at a speed that depends on pool indexes evolution (see theorem 5.2.1), according to the interest rates model of Morpho. The amount of underlying tokens corresponding to an amount in peer-to-peer scaled unit is obtained by multiplying the scaled unit by the peer-to-peer index. They are updated by update_λ^{P2P}.

$$\lambda^{P2P} : \Theta_{M_\psi} \to \mathbb{R}_+$$

We denote by $r^{P2P} : \Theta_{M_\psi} \to \mathbb{R}_+$ the rate at which this index evolves. It is helpful to explain what is the rate experienced by the users of Morpho, but it is not part of the storage. Indeed, Morpho does not use the rates of the pool to compute the indexes. Instead Morpho is only relying on the indexes, which makes it less susceptible to rate manipulations, where an attacker triggers functions on the underlying pool to use the difference of rates it induces. Subsection 5 details the computations and estimation of Morpho's rates.

3.2 Utils functions

Matching functions: These functions match (or unmatch) borrowers or suppliers, updating their balances and returning the amount that was successfully matched (or unmatched). Those functions are described in more detail in subsection 4. They are said internal because they are accessible by Morpho itself only.

Update index: The function $\texttt{update\_}\lambda^M$ updates both the pool indexes and the peer-to-peer indexes of the given market.

$$\texttt{update\_}\lambda^M : \Theta_{M_\psi} \to \mathbb{1}$$

In order to update the pool indexes, this function calls $\texttt{update\_}\mathit{lambda}$. See section 5 for more details on the calculations related to this function.

Borrowing capacity: The function $\texttt{borrowing\_capacity}^M$ is a function that computes and returns a given user's borrowing capacity (in value). This amount can be borrowed in addition to the current debt.

$$\texttt{borrowing\_capacity}^M : A \to \mathbb{R}$$

This function first updates all the indexes it needs to access by calling the function $\texttt{update\_}\lambda^M$ on the respective markets. It then computes its result according to the following formula:

$$\begin{aligned}\texttt{borrowing\_capacity}^M(a) \quad = \quad & \sum_{\theta\in\Theta_{M_\psi}} (s_\theta^{P2P}(a)\cdot\lambda_\theta^{P2P} + s_\theta^{Pool}(a)\cdot\lambda_\theta^{S})\cdot p_\theta\cdot F_\theta \\ & - \sum_{\theta\in\Theta_{M_\psi}} (b_\theta^{P2P}(a)\cdot\lambda_\theta^{P2P} + b_\theta^{Pool}(a)\cdot\lambda_\theta^{B})\cdot p_\theta\end{aligned}$$

3.3 Main Entry Points

Morpho mirrors PLF's five main functions: supply, withdraw, borrow, and repay for users, and liquidate for liquidators. From a user perspective, the experience is the same as on PLFs.

From a high-level logic perspective, supply and borrow do a peer-to-peer matching phase before finding or putting the remaining liquidity on the pool.

Withdraw and repay first try to remove some liquidity from the pool before matching some users on the pool and finally unmatching peer-to-peer matched users. Liquidate is a combination of repay and withdraw. The order of operations has been chosen to maximize the matched peer-to-peer liquidity, given the computation constraints.

3.3.1 Supply

The $\texttt{supply}^M$ function is used to deposit funds to the Morpho protocol, which starts accumulating interest at a potentially improved rate compared to the underlying lending pool.

$$\texttt{supply}^M : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

High-level description

1. **Peer-to-peer supply:**
   (a) **Promote borrowers:** Morpho matches the incoming liquidity with some pool debt. The matching engine, described in more detail in section 4, tries to match as much debt as possible from pool borrowers. The liquidity is used to `repay` their debt on the pool.
2. **Pool supply:**
   (a) **No matching process:** The remaining liquidity of the supplier is deposited on the underlying pool with a call to `supply`. There is no matching process here since the position is not peer-to-peer.

Formal description

```
supply$_\theta^M$(user, amount):
    update_$\lambda_\theta^M$()
    transfer$_\theta$(user, $\overline{M_\psi}$, amount)

    // Peer-to-peer supply

    //// Promote borrowers
```

```
matched = matchBorrowers$_\theta$(amount)
$s_\theta^{P2P}$(user) += matched / $\lambda_\theta^{P2P}$
amount -= matched
repay$_\theta$($\overline{M_\psi}$, matched)

// Pool supply

$s_\theta^{Pool}$(user) += amount / $\lambda_\theta^{S}$
supply$_\theta$($\overline{M_\psi}$, amount)
```

3.3.2 Borrow

The $\texttt{borrow}^M$ function is used to borrow funds from the Morpho protocol, potentially paying less interest compared to the underlying lending pool.

$$\texttt{borrow}^M : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

High-level description

1. **Peer-to-peer borrow:**
   (a) **Promote suppliers:** Morpho matches the incoming demand with supply on the pool. The matching engine tries to match as much supply as possible from pool suppliers. The liquidity is taken from the pool suppliers with a `withdraw` on the underlying pool.
2. **Pool borrow:**
   (a) **No matching process:** The remaining demand is met on the underlying pool with a call to `borrow`.

One may ask how Morpho ensures that its position is fairly collateralized, considering that the borrowers' supply could already be matched peer-to-peer in Morpho: see theorem 3.4.1.

Formal description

```
borrow$_\theta^M$(user, amount):
    update_$\lambda_\theta^M$()
    if amount * $p_\theta$ > borrowing_capacity$^M$(a):
        return
    initialAmount = amount

    // Peer-to-peer borrow

    //// Promote suppliers
    matched = matchSuppliers$_\theta$(amount)
    $b_\theta^{P2P}$(user) += matched / $\lambda_\theta^{P2P}$
    amount -= matched
    withdraw$_\theta$($\overline{M_\psi}$, matched)

    // Pool borrow

    $b_\theta^{Pool}$(user) += amount / $\lambda_\theta^B$
    borrow$_\theta$($\overline{M_\psi}$, amount)

    transfer$_\theta$($\overline{M_\psi}$, user, initialAmount)
```

3.3.3 Withdraw

The `withdraw`^{M} function is used to withdraw previously supplied assets to the Morpho protocol.

$$\texttt{withdraw}^M : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

High-level description

1. **Pool withdraw:**

   (a) **No matching process:** If some of the user's liquidity is supplied on the pool, Morpho withdraws the corresponding part of its position on the underlying pool. It favors this option to maximize the user's matched peer-to-peer liquidity, thus, their capital efficiency. A `withdraw` on the underlying pool is performed.

2. **Peer-to-peer withdraw:**

(a) **Transfer withdraw:**

   i. **Promote suppliers:** If the user is matched peer-to-peer, Morpho replaces him with other pool suppliers. It favors this option before the third one to maximize the total peer-to-peer liquidity. The matching engine tries to match as much liquidity as possible from pool suppliers. The liquidity is retrieved by a call to `withdraw` on the underlying pool.

(b) **Breaking withdraw:**

   i. **Demote borrowers:** Morpho breaks the peer-to-peer credit lines of the withdrawing user with the matched borrowers and reconnects them to the pool. The matching engine unmatches all the remaining debt from peer-to-peer borrowers. A `borrow` on the pool is performed to reconnect the borrowers to the pool. The borrowed assets are transferred to the withdrawer.

Once again, the non-liquidation theorem 3.4.1 ensures that Morpho's position is fairly collateralized.

Formal description

```
withdraw$_\theta^M$(user, amount):
    update_$\lambda_\theta^M$()
    if amount * $F_\theta$ * $p_\theta$ > borrowing_capacity$^M$(user):
        return
    initialAmount = amount

    // Pool withdraw

    withdrawnFromPool = min(amount, $s_\theta^{Pool}$(user) * $\lambda_\theta^S$)
    $s_\theta^{Pool}$(user) -= withdrawnFromPool / $\lambda_\theta^S$
    amount -= withdrawnFromPool
    withdraw$_\theta$($\overline{M_\psi}$, withdrawnFromPool)

    // Peer-to-peer withdraw

    //// Transfer withdraw
```

```
// Promote suppliers
matched = matchSuppliers$_\theta$(amount)
$s_\theta^{P2P}$(user) -= matched / $\lambda_\theta^{P2P}$
amount -= matched
withdraw$_\theta$($\overline{M_\psi}$, matched)

//// Breaking withdraw

// Demote borrowers
unmatched = unmatchBorrowers$_\theta$(amount)
$s_\theta^{P2P}$(user) -= unmatched / $\lambda_\theta^{S}$
borrow$_\theta$($\overline{M_\psi}$, unmatched)

transfer$_\theta$($\overline{M_\psi}$, user, initialAmount)
```

3.3.4 Repay

The $\texttt{repay}^M$ function is used to repay the accumulated debt to the Morpho protocol.

$$\texttt{repay}^M : \Theta_\psi \times A \times \mathbb{R}_+ \to \mathbb{1}$$

High-level description

1. **Pool repay:**

   (a) **No matching process:** If some of the user's debt is borrowed from the pool, Morpho repays liquidity to the underlying pool. It favors this option to maximize the user's matched liquidity and capital efficiency. A `repay` is performed on the underlying pool.

2. **Peer-to-peer repay:**

   (a) **Transfer repay:**

      i. **Promote borrowers:** If the user is matched peer-to-peer, Morpho replaces the peer-to-peer credit lines with other pool borrowers. It favors this option before the third one to maximize the total peer-to-peer liquidity. The matching engine tries to match as much debt as possible from pool borrowers.

A `repay` on the pool is performed to cut those borrowers' debt on the pool.

(b) **Breaking repay:**

i. **Demote suppliers:** Morpho breaks the peer-to-peer credit lines of the repaying user with the remaining matched suppliers and deposits the unmatched liquidity on the underlying pool. The matching engine unmatches all the remaining liquidity from peer-to-peer suppliers. Morpho `supply` on the underlying pool to reconnect the suppliers to the pool.

Formal description

```
repay$_\theta^M$(user, amount):
    update_$\lambda_\theta^M$()
    if amount > $b_\theta^{P2P}$(user) * $\lambda^{P2P}$ + $b^{Pool}$(user) * $\lambda^B$:
        return
    transfer$_\theta$(user, $\overline{M_\psi}$, amount)

    // Pool repay

    repaidOnPool = min(amount, $b_\theta^{Pool}$(user) * $\lambda_\theta^B$)
    $b_\theta^{Pool}$(user) -= repaidOnPool / $\lambda_\theta^B$
    amount -= repaidOnPool
    repay$_\theta$($\overline{M_\psi}$, repaidOnPool)

    // Peer-to-peer repay

    //// Transfer repay

    // Promote borrowers
    matched = matchBorrowers$_\theta$(amount)
    $b_\theta^{P2P}$(user) -= matched / $\lambda_\theta^{P2P}$
    amount -= matched
    repay$_\theta$($\overline{M_\psi}$, matched)

    //// Breaking repay
```

```
// Demote suppliers
unmatched = unmatchSuppliers$_\theta$(amount)
$b_\theta^{P2P}$(user) -= unmatched / $\lambda_\theta^{P2P}$
supply$_\theta$($\overline{M_\psi}$, unmatched)
```

3.3.5 Liquidate

The liquidation on Morpho works the same way as liquidation on the underlying pool. When a position of a user is not sufficiently collateralized, a liquidator can repay part of the debt of the user and seize a part of the collateral of the user.

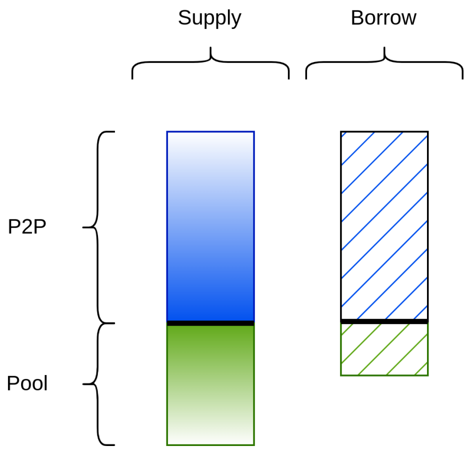
$$\texttt{liquidate}^M : \Theta_\psi \times \Theta_\psi \times A \times A \times \mathbb{R}_+ \to \mathbb{1}$$

Morpho uses the same values for the collateral factors F, `closeFactor`, and `liquidationIncentive` as the ones used by the underlying pool and the same oracle to retrieve the prices of the borrowed and collateralized assets. The main difference with the corresponding function on the pool is the calculation of the underlying amount of supplied and borrowed assets. For each user, both pool and peer-to-peer balances should be accounted. For example, the amount of borrowed assets can be computed by:

```
borrows = $b_{\theta_b}^{Pool}$(user) * $\lambda_{\theta_b}^{B}$ + b$_{\theta_b}^{P2P}$(user) * $\lambda_{\theta_b}^{P2P}$
```

3.4 Non-liquidation theorem

Let us first represent the positions on Morpho with a chart.



The bar on the left represents all the assets that are supplied through Morpho (in an arbitrary common unit), and the bar on the right represents all the assets borrowed through Morpho. In both cases, the bottom part corresponds to the liquidity on the underlying pool because it did not find a peer-to-peer match. Notice that the matched part of the supply equals the matched part of the borrow.

The non-liquidation theorem states that the position of Morpho on the underlying pool is not liquidatable. The intuition behind it comes from considering safe positions as positions where the borrowed assets represent a fraction, in value, of the supplied assets. Then, because Morpho enforces that its users have safe positions (considering both their assets on the pool and their matched assets) thanks to the liquidation mechanism, the combination of pool borrow and peer-to-peer borrow must also represent a fraction of the combination of the pool supply and the peer-to-peer supply. In the previous chart, this is visualized by estimating the whole borrow bar against the whole supply bar. Removing the matched part, both in borrow and supply, only makes this fraction smaller. This amounts to comparing the pool borrow against the pool supply, which gives us the position of Morpho itself on the underlying pool. The chart visualizes this by estimating the bottom of the borrow bar against the bottom of the supply bar.

First, we present and prove the *pool position's integrity* lemma and the *peer-to-peer integrity* lemma, which are helpful to prove the non-liquidation theorem. Recall that b_θ et s_θ are the borrow and supply balances on the underlying pool.

Lemma 3.4.1 (Pool position's integrity)**.** *Morpho's position on the pool is always the sum of the positions on the pool of all its users.*

$$\forall \theta \in \Theta_{M_\psi}, \begin{cases} s_\theta(\overline{M_\psi}) = \sum_u s_\theta^{Pool}(u) \\ b_\theta(\overline{M_\psi}) = \sum_u b_\theta^{Pool}(u) \end{cases}$$

Proof. We prove this lemma by induction.

Base case At deployment, there is no user on Morpho:

$$\sum_u s^{Pool}(u) = \sum_u b^{Pool}(u) = 0$$

And Morpho does not have any position on the pool:

$$s(\overline{M_\psi}) = b(\overline{M_\psi}) = 0$$

So the equalities are verified at the initial state.

Induction step Let $\sigma_i \in \Sigma$ be a state where the equalities are verified. Consider a transition to $\sigma_r \in \Sigma$. We only treat the case where the transition is associated to a function call to `supply`. Let $d = (\theta, u, x) \in D_{\texttt{supply}}$ be the input for this transition. By now, we only consider the market θ, as others are untouched so that θ can be omitted as a subscript.

To remain concise, we write λ^S for $\sigma_r(\lambda^S)$ and λ^B for $\sigma_r(\lambda^B)$. Consider first the **peer-to-peer supply** step. Let x^{μ_B} be the total amount matched in underlying by `matchBorrowers` (`matched` in the pseudo-code). The total pool borrow amount is reduced by x^{μ_B}/λ^B, and the Morpho borrow position on the pool is reduced by x^{μ_B}/λ^B.

$$\sum_u \sigma_r(b^{Pool}(u)) = \sum_u \sigma_i(b^{Pool}(u)) - \frac{x^{\mu_B}}{\lambda^B}$$

$$\sigma_r(b(\overline{M_\psi})) = \sigma_i(b(\overline{M_\psi})) - \frac{x^{\mu_B}}{\lambda^B}$$

Consider also the **pool supply** step. The amount x^{Pool}/λ^S is added to the supplier's pool balance with $x^{Pool} = (x - \texttt{matched})$. The pool supply balance increases by x^{Pool}/λ^S:

$$\sum_u \sigma_r(s^{Pool}(u)) = \sum_u \sigma_i(s^{Pool}(u)) + \frac{x^{Pool}}{\lambda^S}$$

$$\sigma_r(s(\overline{M_\psi})) = \sigma_i(s(\overline{M_\psi})) + \frac{x^{Pool}}{\lambda^S}$$

Yet, the equalities were verified at σ_i, so:

$$\sigma_r(s(\overline{M_\psi})) = \sum_u \sigma_r(s^{Pool}(u))$$

$$\sigma_r(b(\overline{M_\psi})) = \sum_u \sigma_r(b^{Pool}(u))$$

The proof is similar for the other possible transitions. ■

Lemma 3.4.2 (Peer-to-peer integrity)**.** *The sum of the suppliers' peer-to-peer balances is always equal to the sum of the borrowers' peer-to-peer balances.*

$$\forall \theta \in \Theta_{M_\psi}, \sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{P2P} = \sum_u b_\theta^{P2P}(u) \cdot \lambda_\theta^{P2P}$$

Proof. We prove this lemma by induction and reuse the same notations as for lemma 3.4.1.

Base case The equality is clearly verified at deployment.

Induction step Let us prove that the invariant is preserved by a transition resulting from calling the `supply` function. Let $d = (\theta, u, x) \in D_{\texttt{supply}}$ be the input for this function call.

First, we notice that the equality is still verified after the index update:

$$\sum_u s_\theta^{P2P}(u) \cdot \sigma_r(\lambda_\theta^{P2P}) = \sum_u b_\theta^{P2P}(u) \cdot \sigma_r(\lambda_\theta^{P2P})$$

And, to remain concise, we write λ_θ^{P2P} for $\sigma_r(\lambda_\theta^{P2P})$.

Consider first the **peer-to-peer supply** step. Let x^{μ_B} be the total amount matched in underlying by $\texttt{matchBorrowers}_\theta$ (`matched` in the pseudocode). The peer-to-peer supply balance of the supplier is increased by $x^{\mu_B}/\lambda_\theta^{P2P}$, and the total of the peer-to-peer borrow balances is increased by $x^{\mu_B}/\lambda_\theta^{P2P}$.

$$\sum_u \sigma_r(s^{P2P}(u)) \cdot \lambda_\theta^{P2P} = \left(\sum_u \sigma_i(s^{P2P}(u)) + \frac{x^{\mu_B}}{\lambda_\theta^{P2P}} \right) \cdot \lambda_\theta^{P2P}$$

$$\sum_u \sigma_r(b^{P2P}(u)) \cdot \lambda_\theta^{P2P} = \left(\sum_u \sigma_i(b^{P2P}(u)) + \frac{x^{\mu_B}}{\lambda_\theta^{P2P}} \right) \cdot \lambda_\theta^{P2P}$$

Remark also that the **pool supply** step does not change any user peer-to-peer balance. Yet, the equality was verified at σ_i, so:

$$\sum_u \sigma_r(s^{P2P}(u)) \cdot \lambda_\theta^{P2P} = \sum_u \sigma_r(b^{P2P}(u)) \cdot \lambda_\theta^{P2P}$$

The proof is similar for the other possible transitions. ■

We now turn to the non-liquidation theorem itself and its proof. This theorem assumes a working liquidation system on Morpho: positions on Morpho are liquidated as needed. In practice, for this assumption to hold, it requires a reliable, and responsive off-chain monitoring of the positions on Morpho.

Theorem 3.4.1 (Non-liquidation)**.** *If Morpho is equipped with a working liquidation system, then Morpho's position on the pool is not liquidatable.*

Proof. From the peer-to-peer integrity lemma 3.4.2, we know that for each token θ, the following inequality stands:

$$\sum_u b_\theta^{P2P}(u) \cdot \lambda_\theta^{P2P} = \sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{P2P}$$

We also get the following equalities from the pool position's integrity lemma 3.4.1:

$$s_\theta(\overline{M_\psi}) = \sum_u s_\theta^{Pool}(u) \tag{3}$$

$$b_\theta(\overline{M_\psi}) = \sum_u b_\theta^{Pool}(u) \tag{4}$$

Recalling that F_θ represents the collateral factor, the liquidation mechanism on Morpho ensures that, for each user u, we have:

$$\sum_\theta p_\theta \cdot \omega_\theta^B(u) \leq \sum_\theta p_\theta \cdot F_\theta \cdot \omega_\theta^S(u)$$

so

$$\sum_u \sum_\theta p_\theta \cdot \omega_\theta^B(u) \leq \sum_u \sum_\theta p_\theta \cdot F_\theta \cdot \omega_\theta^S(u)$$

and

$$\sum_\theta p_\theta \cdot \sum_u \omega_\theta^B(u) \leq \sum_\theta p_\theta \cdot F_\theta \cdot \sum_u \omega_\theta^S(u)$$

Using the equalities 1, 2 and then the equalities 3 and 4, we get:

$$\begin{aligned}\sum_{\theta} p_{\theta} \cdot (b_{\theta}(\overline{M_{\psi}}) \cdot \lambda_{\theta}^{B} + \sum_{u} b_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P}) \leq \\ \sum_{\theta} p_{\theta} \cdot F_{\theta} \cdot (s_{\theta}(\overline{M_{\psi}}) \cdot \lambda_{\theta}^{S} + \sum_{u} s_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P})\end{aligned} \tag{5}$$

For each θ, F_{θ} less than or equal to 1, so we have:

$$\begin{aligned}\sum_{u} b_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P} - F_{\theta} \cdot \sum_{u} s_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P} &= (1 - F_{\theta}) \cdot \sum_{u} b_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P} \\ &\geq 0\end{aligned}$$

This gives us the following:

$$\sum_{\theta} p_{\theta} \cdot \left[\sum_{u} b_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P} - F_{\theta} \cdot \sum_{u} s_{\theta}^{P2P}(u) \cdot \lambda_{\theta}^{P2P}\right] \geq 0 \tag{6}$$

With 5 and 6, we can conclude that:

$$\sum_{\theta} p_{\theta} \cdot b_{\theta}(\overline{M_{\psi}}) \cdot \lambda_{\theta}^{B} \leq \sum_{\theta} p_{\theta} \cdot F_{\theta} \cdot s_{\theta}(\overline{M_{\psi}}) \cdot \lambda_{\theta}^{S}$$

In other words, Morpho's position on the pool is not liquidatable. ■

4 Matching engine

The role of the matching engine is to create and break the peer-to-peer credit lines of users. In this section, we will first explain how the Morpho protocol makes use of the matching engine, and we will then describe its high-level operation. The implementation is voluntarily left abstract, as different ones could fit into the setting.

4.1 Overview

Let us give an example before describing the inner workings of the matching engine.

Example 4.1.1. *Let us say that the first* 10 *users of Morpho are* `DAI` *suppliers. They all supply* 1000 `DAI` *tokens. Alice deposits some WETH as collateral and borrows* 5000 `DAI`*. At that time, the matching engine will look for suppliers to match and move* 5 *of them peer-to-peer. Alice and* 5 *of the suppliers ended up matched peer-to-peer, enjoying Morpho's rates.*

Note that it is possible for a user to be only partially matched peer-to-peer. Such a user has a fraction of its liquidity matched peer-to-peer, while the rest is on the pool.

Notice, moreover, that peer-to-peer credit lines are not really associating borrowers with suppliers. Being matched peer-to-peer is a state in which the user benefits from better rates than on the pool. And the matching engine's role is to move users in and out of this state.

4.2 Storage

User addresses are kept in four structures. Suppliers are in `suppliersInP2P` if they have matched supply and in `suppliersOnPool` if they have some supply on the pool. The same applies to borrowers, with `borrowersInP2P` and `borrowersOnPool`.

In this yellow paper, we do not describe the data structures used for the storage of the matching engine. We assume that those structures are always up to date and that we have access to functions to retrieve the suppliers and borrowers that are in those structures.

4.3 Functions

4.3.1 Getter functions

Get a peer-to-peer supplier: This function returns a supplier that has matched peer-to-peer liquidity in the market θ.

$$\texttt{getSupplier}^{P2P} : \Theta_{M_\psi} \to A$$

Get a pool supplier: This function returns a supplier that has liquidity on the pool in the market θ.

$$\texttt{getSupplier}^{Pool} : \Theta_{M_\psi} \to A$$

Get a peer-to-peer borrower: This function returns a borrower that has matched peer-to-peer debt in the market θ.

$$\texttt{getBorrower}^{P2P} : \Theta_{M_\psi} \to A$$

Get a pool borrower: This function returns a borrower that has debt on the pool in the market θ.

$$\texttt{getBorrower}^{Pool} : \Theta_{M_\psi} \to A$$

4.3.2 Matching functions

Let us describe the main functions of the matching engine: `matchSuppliers`, `unmatchSuppliers`, `matchBorrowers`, `unmatchBorrowers`. Those functions match (resp. unmatch) peer-to-peer borrowers or peer-to-peer suppliers by looking for them in the corresponding structure of the matching engine. They return the amount that was successfully matched (resp. unmatched), and we assume that they update the structures of the matching engine accordingly.

Match suppliers: The function `matchSuppliers` matches pool suppliers until the amount `amount` has been found or until there are no more pool suppliers.

$$\texttt{matchSuppliers} : \Theta_{M_\psi} \times \mathbb{R}_+ \to \mathbb{R}_+$$

It updates some suppliers' positions in the structures of the matching engine (`suppliersInP2P` and `suppliersOnPool`) and their balances (s^{P2P}, s^{Pool}) accordingly. Then it returns the matched amount.

As an example, the following pseudo-code loops through pool suppliers to find the asked liquidity.

```
matchSuppliers$_\theta$(amount):
    matched = 0
    while amount > 0 and suppliersOnPool != $\emptyset$:
        supplier = getSupplier$_\theta^{Pool}$()
        toMatch = min(amount, $s_\theta^{Pool}$(supplier) * $\lambda_\theta^S$)
        $s_\theta^{Pool}$(supplier) -= toMatch / $\lambda_\theta^S$
        $s_\theta^{P2P}$(supplier) += toMatch / $\lambda_\theta^{P2P}$
        amount -= toMatch
        matched += toMatch
    return matched
```

Remember that the data-structure `suppliersOnPool` is assumed to always be up to date so we can expect the loop to terminate if there are no more pool suppliers.

Unmatch suppliers: The function `unmatchSuppliers` unmatches peer-to-peer suppliers, until the amount `amount` has been found or until there are no more peer-to-peer suppliers.

$$\texttt{unmatchSuppliers} : \Theta_{M_\psi} \times \mathbb{R}_+ \to \mathbb{R}_+$$

It updates some suppliers' positions in the structures of the matching engine (`suppliersInP2P` and `suppliersOnPool`) and their balances (s^{P2P}, s^{Pool}) accordingly. Then it returns the unmatched amount.

Match borrowers: The function `matchBorrowers` matches pool borrowers, until the amount `amount` has been found or until there are no more pool borrowers.

$$\texttt{matchBorrowers} : \Theta_{M_\psi} \times \mathbb{R}_+ \to \mathbb{R}_+$$

It updates some borrowers' positions in the structures of the matching engine (`borrowersInP2P` and `borrowersOnPool`) and their balances (b^{P2P}, b^{Pool}) accordingly. Then it returns the matched amount.

Unmatch borrowers: The function `unmatchBorrowers` unmatches peer-to-peer borrowers, until the amount `amount` has been found or until there are no more peer-to-peer borrowers.

$$\texttt{unmatchBorrowers} : \Theta_{M_\psi} \times \mathbb{R}_+ \to \mathbb{R}_+$$

It updates some borrowers' positions in the structures of the matching engine (`borrowersInP2P` and `borrowersOnPool`) and their balances (b^{P2P}, b^{Pool}) accordingly. Then it returns the unmatched amount.

5 Interest rates mechanism

In this section, we will go through all the Morpho mechanisms related to interest rates. To this end, we first define the rates that we will consider and then explain the different methods used to update the indexes, including Morpho's approach. We conclude the section with a proof of the rate improvement that Morpho is achieving.

5.1 Interest rates

5.1.1 Pool rates

Morpho's users that are not matched peer-to-peer are connected to the pool (either via a deposit or a loan) and so experience pool rates. Their credit lines evolve at pool rates, r^S and r^B.

5.1.2 Peer-to-peer theoretical rate

While users on the pool necessarily earn and pay the exact pool rates, Morpho can freely choose the interest rates of its peer-to-peer matched users. In order for the protocol to be a Pareto improvement of liquidity pools, the chosen rate should always be in the rate spread of the pool.

For this first version of the protocol, it is a weighted arithmetic mean of the supply and borrow rates of the pool, weighted by $\alpha \in [0, 1]$, the *peer-to-peer cursor*. The corresponding peer-to-peer rate is:

$$\begin{aligned} r^\alpha &= r^S + \alpha(r^B - r^S) \\ &= (1 - \alpha)r^S + \alpha r^B \end{aligned}$$

For example, the so-called "mid-rate" ($\alpha = \frac{1}{2}$) would be:

$$r^\alpha = r^{P2P} = \frac{r^S + r^B}{2}$$

5.1.3 Peer-to-peer reserve rate

In order to be sustainable in the long run, the protocol can charge a fee for users and give it back to the Morpho DAO. In this section we focus on the impact the fee has on Morpho's rates, an explanation of the fee's accounting is given in section 6.2.4. The fee is only charged for users who really benefit from the protocol improvements, so peer-to-peer matched users[15]. Thus, a difference between matched suppliers' and borrowers' rates is introduced. Call ρ the *reserve factor*, a quantity that represents the proportion of the difference with the pool rate. The resulting rates are:

$$\begin{aligned} r^{\rho^S} &= r^\alpha - \rho(r^\alpha - r^S) \\ &= (1-\rho)r^\alpha + \rho r^S \\ &= (1-\rho)((1-\alpha)r^S + \alpha r^B) + \rho r^S \\ &= (1-\alpha+\rho\alpha)r^S + (\alpha - \rho\alpha)r^B \end{aligned}$$

$$\begin{aligned} r^{\rho^B} &= r^\alpha + \rho(r^B - r^\alpha) \\ &= (1-\rho)r^\alpha + \rho r^B \\ &= (1-\rho)((1-\alpha)r^S + \alpha r^B) + \rho r^B \\ &= (1-\rho-\alpha+\rho\alpha)r^S + (\rho+\alpha-\rho\alpha)r^B \end{aligned}$$

The difference $r^{\rho^B} - r^{\rho^S} = \rho(r^B - r^s)$ is the percentage ρ of the spread $r^B - r^S$ of the pool rates.

5.2 Indexes

5.2.1 Generalities

Let $(r_i)_{i\in\mathbb{N}}$ be a sequence of rates and $(\lambda_i)_{i\in\mathbb{N}}$ be the corresponding sequence of indexes. Let $(t_i)_{i\in\mathbb{N}}$ be the increasing sequence of times when the index is updated. We assume that $t_0 = 0$. Let $t_{current}$ be an update time with $current > 0$, we write $t_{last} = t_{current-1}$ and $\Delta t = t_{current} - t_{last}$. We detail the indexes evolution in the following three different settings, with the sequence (λ_i) being either (λ_i^{lin}), (λ_i^{comp}) or (λ_i^{recomp}).

With linear interests. The formula of index evolution in one step is given by:

$$\forall t \in \mathbb{N}, \lambda_{t+1}^{lin} = \lambda_t^{lin} + \lambda_0^{lin} \cdot r_t$$

[15] Thus, the fee can be seen as a cut in the improvement that Morpho enables

By recurrence, we can prove that:

$$\forall t \in \mathbb{N}, \lambda_t^{lin} = \lambda_0^{lin}(1 + \sum_{k<t} r_k)$$

Assuming that (r_i) is constant on $[\![t_{last}, t_{current}[\![$, the following formula gives a way to update the indexes in the case of linear interests:

$$\lambda_{t_{current}}^{lin} = \lambda_{t_{last}}^{lin} + \Delta t \cdot \lambda_0^{lin} \cdot r_{t_{last}}$$

With compounded interests.[16] The formula of index evolution in one step is given by:

$$\forall t \in \mathbb{N}, \lambda_{t+1}^{comp} = \lambda_t^{comp} + \lambda_t^{comp} \cdot r_t$$

By recurrence, we can prove that:

$$\forall t \in \mathbb{N}, \forall t' \in \mathbb{N}, t' < t, \lambda_t^{comp} = \lambda_{t'}^{comp} \prod_{t' \leq i < t} (1 + r_i)$$

Assuming that (r_i) is constant on $[\![t_{last}, t_{current}[\![$[17], the following formula gives a way to update the index in the case of compounded interests:

$$\lambda_{t_{current}}^{comp} = \lambda_{t_{last}}^{comp}(1 + r_{t_{last}})^{\Delta t}$$

With re-compounded linear interests.[18]**:** The formula of index evolution in one step is given by:

$$\forall t \in \mathbb{N}, \lambda_{t+1}^{recomp} = \lambda_t^{recomp} + \lambda_{t'}^{recomp} \cdot r_t$$

where t' is the last update time before t. By recurrence, we can prove that, for $s, e \in \mathbb{N}$ such that $s \leq e$:

$$\lambda_{t_e}^{recomp} = \lambda_{t_s}^{recomp} \prod_{s \leq i < e} \left(1 + \sum_{t_i \leq j < t_{i+1}} r_j\right)$$

And, in particular:

$$\lambda_{t_{current}}^{recomp} = \lambda_{t_{last}}^{recomp} \left(1 + \sum_{t_{last} \leq j < t_{current}} r_j\right)$$

[16] Aave uses an approximation of this method for borrow interests.

[17] This is the case in reality because indexes are updated at each pool interaction, and rates only change at pool interactions.

[18] This is the method used by Compound and for supply interest by Aave.

Assuming that (r_j) is constant on $[\![t_{last}, t_{current}[\![$, the following formula gives a way to update the index in the case of re-compounded interests:

$$\lambda_{t_{current}}^{recomp} = \lambda_{t_{last}}^{recomp}(1 + \Delta t \cdot r_{t_{last}})$$

5.2.2 Indexes calculations on Morpho

Let us define the weights β^S and β^B, with $\beta^B + \beta^S = 1$ such that the Morpho rate r is expected to be $\beta^S r^S + \beta^B r^B$. These weights define where the Morpho rate lies in the spread. For example, if β^S is close to 1, r should be close to the pool supply rate r^S. λ denotes the index associated to r, allowing us to generalize the results of this subsection. In particular, we will be able to apply the theorem 5.2.1 to the rates r^α, r^{ρ^S}, and r^{ρ^B} defined in subsection 5.1 and to the rates r^{γ^S} and r^{γ^B} defined in subsection 6.3.

As previously stated, Morpho does not use the pool rates to update its indexes but instead uses the pool indexes' growth. The following theorem gives the formula that is used to update Morpho's indexes in that way. Notice that the pool rates do not appear in the formula given by the theorem.

We assume that the underlying pool uses compounded interests to update its indexes.

Theorem 5.2.1. *Let $\beta^S \in \mathbb{R}_+$, $\beta^B \in \mathbb{R}_+$ with $\beta^S + \beta^B = 1$ and let $(\lambda_i)_{i\in\mathbb{N}}$ be a sequence such that:*

$$\forall t \in \mathbb{N}, \forall t' \in \mathbb{N}, t > t' \Rightarrow \lambda_t = \lambda_{t'} \prod_{t' \leq i < t} (1 + \beta^S r_i^S + \beta^B r_i^B)$$

We have, with $R_{t',t}$ the maximum of the rates r_i^B for $i \in [\![t', t[\![$:

$$\forall t \in \mathbb{N}, \forall t' \in \mathbb{N}, t > t' \Rightarrow \lambda_t = \lambda_{t'} \left(\beta^S \frac{\lambda_t^S}{\lambda_{t'}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{t'}^B} + O(R_{t',t}^2) \right)$$

Proof. Let $t \in \mathbb{N}, t' \in \mathbb{N}, t' < t$. The precision of the approximation given by this theorem relies on the fact that the rates are small numbers, and thus the maximum of those rates $R_{t',t}$ is small too.

$$
\begin{aligned}
\lambda_t &= \lambda_{t'} \prod_{t' \le i < t} (1 + \beta^S r_i^S + \beta^B r_i^B) \\
&= \lambda_{t'} \left(1 + \sum_{t' \le i < t} (\beta^S r_i^S + \beta^B r_i^B) + O(R_{t',t}^2) \right) \\
&= \lambda_{t'} \left(1 + \beta^S \sum_{t' \le i < t} r_i^S + \beta^B \sum_{t' \le i < t} r_i^B + O(R_{t',t}^2) \right)
\end{aligned}
$$

On the other hand, we have:

$$
\begin{aligned}
\frac{\lambda_t^S}{\lambda_{t'}^S} &= \frac{\lambda_0 \prod_{0 \le i < t} (1 + r_i^S)}{\lambda_0 \prod_{0 \le i < t'} (1 + r_i^S)} \\
&= \prod_{t' \le i < t} (1 + r_i^S) \\
&= 1 + \sum_{t' \le i < t} r_i^S + O(R_{t',t}^2)
\end{aligned}
$$

and

$$
\begin{aligned}
\frac{\lambda_t^B}{\lambda_{t'}^B} &= \frac{\lambda_0 \prod_{0 \le i < t} (1 + r_i^B)}{\lambda_0 \prod_{0 \le i < t'} (1 + r_i^B)} \\
&= \prod_{t' \le i < t} (1 + r_i^B) \\
&= 1 + \sum_{t' \le i < t} r_i^B + O(R_{t',t}^2)
\end{aligned}
$$

so

$$
\begin{aligned}
\lambda_t &= \lambda_{t'} \left(1 + \beta^S \left(\frac{\lambda_t^S}{\lambda_{t'}^S} - 1 + O(R_{t',t}^2)\right) + \beta^B \left(\frac{\lambda_t^B}{\lambda_{t'}^B} - 1 + O(R_{t',t}^2)\right) + O(R_{t',t}^2) \right) \\
&= \lambda_{t'} \left(1 + \beta^S \frac{\lambda_t^S}{\lambda_{t'}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{t'}^B} - \beta^S - \beta^B + O(R_{t',t}^2) \right) \\
&= \lambda_{t'} \left(\beta^S \frac{\lambda_t^S}{\lambda_{t'}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{t'}^B} + O(R_{t',t}^2) \right)
\end{aligned}
$$

■

Notice that if the underlying pool uses linear interests, then the approximation given by the theorem is an exact calculation. Let us now explain how the indexes are updated in Morpho, starting with the most straighforward case where there is no reserve factor.

Without reserve factor: Let $(u_i)_{i\in\mathbb{N}}$ be the sequences of times when the index is updated on Morpho. Let $(\lambda_i^S)_{i\in\mathbb{N}}$ and $(\lambda_i^B)_{i\in\mathbb{N}}$ be the supply and borrow indexes on the pool. Let $(\lambda_i^\alpha)_{i\in\mathbb{N}}$ be Morpho's indexes. This notation is introduced to take into account the peer-to-peer cursor α.

Let $t \in \mathbb{N}, t' \in \mathbb{N}$ with $t' < t$. We have:

$$\begin{aligned}
\lambda_t^\alpha &= \lambda_{t'}^\alpha \prod_{t'\leq i<t} (1 + r_i^\alpha) \\
&= \lambda_{t'}^\alpha \prod_{t'\leq i<t} (1 + (1-\alpha) r_i^S + \alpha r_i^B)
\end{aligned}$$

With theorem 5.2.1 ($\beta^S = (1-\alpha)$, $\beta^B = \alpha$):

$$\begin{aligned}
\lambda_t^\alpha &= \lambda_{t'}^\alpha \left((1-\alpha) \frac{\lambda_t^S}{\lambda_{t'}^S} + \alpha \frac{\lambda_t^B}{\lambda_{t'}^B} + O(R_{t',t}^2) \right) \\
&\approx \lambda^\alpha \left((1-\alpha) \frac{\lambda_t^S}{\lambda_{t'}^S} + \alpha \frac{\lambda_t^B}{\lambda_{t'}^B} \right)
\end{aligned}$$

This last formula can be used to update the indexes.

With reserve factor: Now we take into account the reserve factor. As we have seen in paragraph 5.1.3, we want the rates experienced by users to be:

$$\begin{aligned}
r^{\rho^S} &= (1 - \alpha + \rho\alpha) r^S + (\alpha - \rho\alpha) r^B \\
r^{\rho^B} &= (1 - \rho - \alpha + \rho\alpha) r^S + (\rho + \alpha - \rho\alpha) r^B
\end{aligned}$$

We denote the corresponding index rates by $(\lambda_i^{\rho^S})_{i\in\mathbb{N}}$ and $(\lambda_i^{\rho^B})_{i\in\mathbb{N}}$. This notation is introduced to take into account the reserve factor (in addition to the peer-to-peer cursor).

Let $t \in \mathbb{N}, t' \in \mathbb{N}$ with $t' < t$. We have:

$$\begin{aligned}
\lambda_t^{\rho^S} &= \lambda_{t'}^{\rho^S} \prod_{t'\leq i<t} (1 + r_i^{\rho^S}) \\
&= \lambda_{t'}^{\rho^S} \prod_{t'\leq i<t} (1 + (1 - \alpha + \rho\alpha) r_i^S + (\alpha - \rho\alpha) r_i^B)
\end{aligned}$$

With theorem 5.2.1 ($\beta^S = (1-\alpha+\rho\alpha)$, $\beta^B = (\alpha - \rho\alpha)$):

$$\lambda_t^{\rho^S} = \lambda_{t'}^{\rho^S}\left((1-\alpha+\rho\alpha)\frac{\lambda_t^S}{\lambda_{t'}^S} + (\alpha-\rho\alpha)\frac{\lambda_t^B}{\lambda_{t'}^B} + O(R_{t',t}^2)\right) \tag{7}$$

Similarly, for the borrow indexes, we have:

$$\lambda_t^{\rho^B} = \lambda_{t'}^{\rho^B}\left((1-\rho-\alpha+\rho\alpha)\frac{\lambda_t^S}{\lambda_{t'}^S} + (\rho+\alpha-\rho\alpha)\frac{\lambda_t^B}{\lambda_{t'}^B} + O(R_{t',t}^2)\right) \tag{8}$$

When considering the reserve factor, we can use those two results to update the indexes. Those formulas are efficient computations that we can use in the code.

5.3 Peer-to-peer inequality

Notice that the *peer-to-peer integrity* lemma no longer holds because of the reserve factor. However, we can still prove that the sum of the suppliers' peer-to-peer balances is always equal to the sum of the borrowers' peer-to-peer balances plus the unpaid reserve in underlying (which we will note P).

Lemma 5.3.1.

$$\forall \theta \in \Theta, \sum_u s^{P2P}(u) \cdot \lambda_\theta^{\rho^S} + P_\theta = \sum_u b^{P2P}(u) \cdot \lambda_\theta^{\rho^B}$$

We have as a direct corrolary:

Lemma 5.3.2. *(Peer-to-peer inequality)*

$$\forall \theta \in \Theta, \sum_u s^{P2P}(u) \cdot \lambda_\theta^{\rho^S} \leq \sum_u b^{P2P}(u) \cdot \lambda_\theta^{\rho^B}$$

This is enough to prove the non-liquidation theorem (3.4.1).

5.4 Rates calculations and Pareto improvement

We show in this subsection that Morpho rates are improved compared to the ones on the underlying pool. To that end, we assume that the initial peer-to-peer index rates are between the supply index rate λ^S and the borrow index rate λ^B and show that index rates stay in this range. An exact calculation of the rates is provided, leading to an estimate of those rates that is easier to understand and compute.

Let (λ_i) be a sequence only changing at (u_n) (those represent the update times) such that, for $t = u_n$, we have:

$$\lambda_t = \lambda_{u_{n-1}} \left(\beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B} \right)$$

with $\beta^S + \beta^B = 1$. This condition holds for (λ_i^α), $(\lambda_i^{\rho^S})$, $(\lambda_i^{\rho^B})$, and even (λ_i^S) and (λ_i^B). Remember that the weights β^S and β^B can be computed from the peer-to-peer cursor α and the reserve factor ρ of Morpho.

5.4.1 Index rates improvement

Theorem 5.4.1. *The index λ_i stays in the interval $[\lambda_i^S, \lambda_i^B]$.*

Proof. We prove this theorem by induction. We assume that it holds at 0, so we only need to prove that it holds at $t = u_n$, knowing that:

$$\lambda_{u_{n-1}}^S \leq \lambda_{u_{n-1}} \leq \lambda_{u_{n-1}}^B$$

Since the supply rate is less than the borrow rate, we have

$$\frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} \leq \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$$

The average of these rates should be in the middle, so:

$$\frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} \leq \beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B} \leq \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$$

By using the induction hypothesis, we get

$$\lambda_{u_{n-1}}^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} \leq \lambda_{u_{n-1}} (\beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}) \leq \lambda_{u_{n-1}}^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$$

which is what we wanted to prove:

$$\lambda_t^S \leq \lambda_t \leq \lambda_t^B$$

■

5.4.2 Rates improvement

Let u_{n-1} be the last time the index rates have been updated. With $\Delta t = t - u_{n-1}$, let $\bar{r}_t^S$ be the rate such that:

$$(1+\bar{r}_t^S)^{\Delta t} = \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S}$$

The rate $\bar{r}_t^S$ is formally defined by:

$$\bar{r}_t^S = \left(\frac{\lambda_t^S}{\lambda_{u_{n-1}}^S}\right)^{1/\Delta t} - 1$$

The rate $\bar{r}_t^S$ is an "average", meaning that it is the rate at which the index rate λ^S would grow if its rate were constant between u_{n-1} and t. In other words, we can consider that a supplier has benefited from this average rate for the whole period Δt. We define $\bar{r}_t^S$ and $\bar{r}_t$ in the same way:

$$(1+\bar{r}_t^B)^{\Delta t} = \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$$

$$(1+\bar{r}_t)^{\Delta t} = \beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$$

So, the rate experienced by matched users of Morpho between the update times u_{n-1} and $t = u_n$ is given by:

$$\bar{r}_t = (\beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B})^{1/\Delta t} - 1$$

Theorem 5.4.2. *The experienced rate $\bar{r}_t$ is in the spread $[\bar{r}_t^S, \bar{r}_t^B]$.*

Proof. Define $f : \mathbb{R}^+ \to \mathbb{R}$ such that for $x \geq 0$, $f(x) = (1+x)^{1/\Delta t} - 1$. We know that

$$\frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} \leq \beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B} \leq \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$$

and since f is growing, we have:

$$f(\frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} - 1) \leq f(\beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B} - 1) \leq f(\frac{\lambda_t^B}{\lambda_{u_{n-1}}^B} - 1)$$

so

$$\bar{r}_t^S \leq \bar{r}_t \leq \bar{r}_t^B$$

■

This shows that Morpho provides better rates than the underlying pool.

5.4.3 Morpho rate calculation and estimate

We want to know where the average rate lies between the supply and the borrow rates. This amounts to calculating β_r^S and β_r^B such that $\beta_r^S + \beta_r^B = 1$ and such that $\bar{r}_t = \beta_r^S \bar{r}_t^S + \beta_r^B \bar{r}_t^B$. Thus, the exact calculation is given by:

$$\beta_r^S = \frac{\bar{r}_t - \bar{r}_t^B}{\bar{r}_t^S - \bar{r}_t^B}$$

and by $\beta_r^B = 1 - \beta_r^S$. This way, any user with access to the underlying pool rates can calculate where the Morpho rate lies between the supply and the borrow rates. If the quantities $\frac{\lambda_t^S}{\lambda_{u_{n-1}}^S}$ and $\frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}$ are small, we can use the following estimation of the function f for small values of x

$$\begin{aligned} f(x) &\approx f(0) + f'(0) \cdot x \\ &\approx \Delta t \cdot x \end{aligned}$$

and estimate

$$\begin{aligned} \beta_r^S &= \frac{\bar{r}_t - \bar{r}_t^B}{\bar{r}_t^S - \bar{r}_t^B} \\ &\approx \frac{\Delta t \cdot (\beta^S \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} + \beta^B \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}) - \Delta t \cdot \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}}{\Delta t \cdot \frac{\lambda_t^S}{\lambda_{u_{n-1}}^S} - \Delta t \cdot \frac{\lambda_t^B}{\lambda_{u_{n-1}}^B}} \\ &\approx \beta^S \end{aligned}$$

We also have $\beta_r^B \approx \beta^B$. Notice that this approximation is independent of time. This gives a simple rule of thumb to know where the Morpho rate lies between the borrow rate and the supply rate.

We have seen that the peer-to-peer rate can be computed precisely or estimated given the appropriate weights β^B (and β^S). Notably, for the peer-to-peer supply rate taking into account the reserve factor, we have $\beta^B = \alpha - \rho\alpha$. This is useful to compute the experienced rate of a user with both a position on the pool and in peer-to-peer. The rate experienced by the user is thus an average of the computed peer-to-peer rate and the pool rate (either r^B or r^S), weighted by their pool and peer-to-peer balances.

6 Peer-to-peer delta mechanism

Matching and unmatching users peer-to-peer is not a constant time operation, and large transactions may not find sufficient liquidity given the alloted resources. The delta mechanism is designed to solve this issue.

Definining a concrete limit on the computation can be done in different settings: it could be done to save CPU time, to limit the total power used, or to prove that a particular implementation terminates by first adding a ”fuel parameter”. Using this limit, we can then be decide at any point if the matching engine should stop. In this paper we apply the delta mechanism to limit the gas usage in the EVM [B+14], but the same mechanism could be used in the contexts mentioned above.

This section first gives a high-level description of the problem and of the delta mechanism as a solution, which allows us to derive the impact it has on the rates. We then turn to a more detailed description of how this is implemented, and we conclude by updating the protocol formalism to take into account the delta mechanism.

6.1 Description

Finding a match can be an intensive computation, and we explain here how we can stop the matching engine early to mitigate its cost. It amounts to finding the missing liquidity when the matching engine did not return a high enough matched amount.

When supplying or borrowing, if not enough liquidity could be found, we already have the pool fallback: Morpho fills the rest of the order with a supply or borrow on the pool, and the user balances are updated accordingly.

For withdrawing and repaying, since these functions end with a matching part that is expected to find the asked liquidity, we need to come up with another fallback. Indeed, we would need to modify the balance of the unmatched users and the data-structures states, but we cannot because of the computation constraint. The idea is to continue the breaking step (see breaking withdraw and breaking repay) by increasing the position of Morpho on the pool accordingly. This means that the rate experienced by suppliers should decrease in case of a breaking repay, and the rate experienced by borrowers should increase in case of a breaking withdraw. The delta mechanism is designed to account for this change.

To summarize, the peer-to-peer delta mechanism works as follows: we operate the pool fallback to fill the order, even if we cannot unmatch enough peer-to-peer users. It introduces a difference[19] between the peer-to-peer liquidity evolving at Morpho's rates and the sum of peer-to-peer balances of users on this side of the market. In order to remedy this, the rates experienced by all peer-to-peer users are adapted.

Concretely, consider the case where one user is withdrawing (resp. repaying), and its liquidity is matched peer-to-peer. Suppose also that there is insufficient liquidity from suppliers (resp. borrowers) on the pool to replace him. So the peer-to-peer credit lines will be broken, and peer-to-peer borrowers (resp. suppliers) will be reconnected to the pool. If the matching engine cannot find enough liquidity in this step, then Morpho borrows (resp. supplies) all the remaining asked liquidity on the pool. This means that all peer-to-peer borrowers (resp. suppliers) now have a part of their borrow (resp. supply) on the pool and now get a worse rate.

Let S^{P2P} and S^{Pool} be the total supply of Morpho users peer-to-peer and on the pool:

$$\begin{aligned} S^{P2P} &= \sum_{a \in A} s^{P2P}(a) \\ S^{Pool} &= \sum_{a \in A} s^{Pool}(a) \end{aligned}$$

Similarly, let B^{P2P} and B^{Pool} be the total borrows:

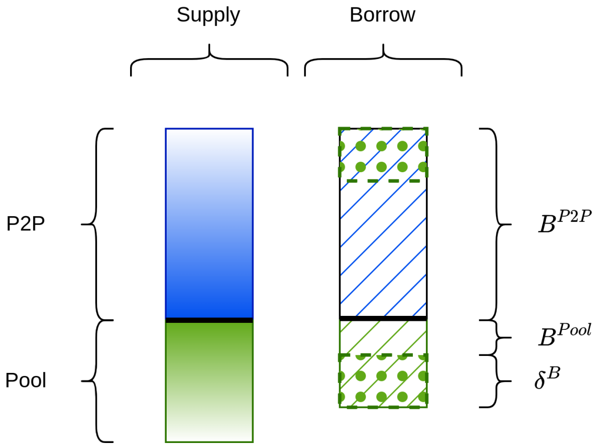
$$\begin{aligned} B^{P2P} &= \sum_{a \in A} b^{P2P}(a) \\ B^{Pool} &= \sum_{a \in A} b^{Pool}(a) \end{aligned}$$

Recall that $s(a)$ and $b(a)$ represent the supply and borrow balances of user a on the pool. Those quantities are defined independently of Morpho in subsection 2.2. The peer-to-peer supply delta δ^S and the peer-to-peer borrow delta δ^B are defined as follows:

$$\begin{aligned} S^{Pool} + \delta^S &= s(\overline{M_\psi}) \\ B^{Pool} + \delta^B &= b(\overline{M_\psi}) \end{aligned}$$

[19] The name ”delta” refers to this difference.

These notations are represented on the following chart, showing only the case of the borrow delta for concision.



Example 6.1.1. *Let us take a simple example. Suppose that the matching engine can unmatch at most three users. The following three successive situations give us an example of the delta mechanism in functioning.*

Suppliers	on pool	in p2p
user 1	0	60

Morpho's position on the pool: 0

Borrowers	on pool	in p2p
user 2	0	10
user 3	0	10
user 4	0	10
user 5	0	10
user 6	0	10
user 7	0	10

Morpho's position on the pool: 0

Table 1: Initial situation

Suppliers	on pool	in p2p
user 1	0	30

Morpho's position on the pool: 0

Borrowers	on pool	in p2p
user 2	10	0
user 3	10	0
user 4	10	0
user 5	0	10
user 6	0	10
user 7	0	10

Morpho's position on the pool: 30

Table 2: After classic breaking withdraw (user 1 withdraws 60)

Suppliers	on pool	in p2p
user 1	0	0

Morpho's position on the pool: 0

Borrowers	on pool	in p2p
user 2	10	0
user 3	10	0
user 4	10	0
user 5	0	10
user 6	0	10
user 7	0	10

Morpho's position on the pool: 60

Table 3: After the whole withdraw

After the whole transaction, the peer-to-peer borrow delta is equal to 30. All the peer-to-peer amount is in effect on the pool.

There are three situations in which Morpho deals with the delta mechanism.

Delta increase After a breaking withdraw or breaking repay, if not enough peer-to-peer users could be unmatched, we increase the delta by the missing liquidity.

Delta matching When new liquidity comes (in a supply or borrow), the delta of the other side of the market is "matched", reducing it.

Delta reduction When some peer-to-peer liquidity exits, the corresponding delta is reduced. For example, withdrawing peer-to-peer matched liquidity will first decrease the supply delta.

6.2 Implementation

To ensure that transactions gas costs are bounded, we make sure that the gas used for matching does not exceed a predefined limit. The predefined limit, written `maxGasForMatching`, accounts for all the gas used by the matching engine during one transaction. When a matching function does not have enough gas left to execute, it can return early and return a matched amount lower than the asked liquidity. Since other parts of the code also have a bounded gas cost, we know that Morpho's operations' overall gas costs are bounded. This way, Morpho can scale as more users enter the different markets.

The matching engine is not mandatory for operations to succeed. Thus we introduce the possibility for Morpho users to choose the `maxGasForMatching` parameter, that is how much gas they are ready to pay for matching engine in $\texttt{supply}^M$ and $\texttt{borrow}^M$ functions.

This cannot be used in $\texttt{withdraw}^M$, $\texttt{repay}^M$, and $\texttt{liquidate}^M$. Indeed, for these functions, there is no economic incentive to match and unmatch more users (in fact, there is an incentive not to do so because of the gas cost). So, for these functions, the maximum gas to be used in matching is fixed. It may be 0 for $\texttt{liquidate}^M$ to reduce liquidations gas cost.

The matching functions implementations need to be modified to consider the gas constraints so that they stop executing when there is no gas left for matching. More specifically, the matching functions described in section 4 now take an additional parameter and return an additional value: respectively, the gas available for the matching engine before the call of the function and the gas left after its execution. Consider, for example, the function `matchSuppliers`, updated to take into account the gas limits

$$\texttt{matchSuppliers} : \Theta_{M_p} \times \mathbb{R}_+ \times \mathbb{N} \to \mathbb{R}_+ \times \mathbb{N}$$

A call such as $\texttt{matchSuppliers}_\theta\texttt{(amount, gasAtStart)}$ will match suppliers and return a pair `(matched, gasLeft)`, where `matched` is the liquidity that has been matched, and `gasLeft` is the gas left after the call, a lower value than `gasAtStart`. Notice that the gas parameter is simply an integer here, as in Ethereum. As before, `matched` is lower than `amount`, the asked liquidity. The matched amount can also be lower than `amount` when there is not enough gas to look for the asked liquidity.

We also need to track the deltas in new storage variables:

$$\delta^S \in \Theta_\psi \to \mathbb{R}_+$$
$$\delta^B \in \Theta_\psi \to \mathbb{R}_+$$

for respectively the peer-to-peer supply delta and the peer-to-peer borrow delta. Their values are in pool scaled units because they evolve at pool rates. Also, to calculate the share of the delta and update the indexes, we need to track the total amounts matched peer-to-peer. To do so, we introduce two new variables to Morpho's contract storage:

$$S^{P2P} \in \Theta_\psi \to \mathbb{R}_+$$

for the total peer-to-peer supply amount and

$$B^{P2P} \in \Theta_\psi \to \mathbb{R}_+$$

for the total peer-to-peer borrow amount. Finally, we define the peer-to-peer supply index that takes into account the delta mechanism (on top of the index cursor and the reserve factor):

$$\lambda^{\gamma^S}$$

Similarly, we have the peer-to-peer borrow index taking into account the delta mechanism:

$$\lambda^{\gamma^B}$$

Subsection 6.3 details how those peer-to-peer indexes are updated.

Let us rewrite Morpho's main functions with the implementation of the delta mechanism.

6.2.1 Supply

High-level description

1. **Peer-to-peer supply:**
   (a) **Match peer-to-peer borrow delta:** Morpho "matches" the user with the peer-to-peer borrow delta and reduces it. This is done first to keep the delta as low as possible. A `repay` on the pool is performed to reduce the surplus of borrow on the pool induced by the peer-to-peer borrow delta. Note that this allows responsive users to skip the queue of users on the pool and be instantly matched peer-to-peer.

   (b) **Peer-to-peer supply:** Morpho matches the incoming liquidity with some debt on the pool. The matching engine tries to match as much debt as possible from pool borrowers. A `repay` on the pool is performed to disconnect the matched borrowers from it.
2. **Pool supply:**
   (a) **No matching process:** The remaining liquidity is deposited on the underlying pool. There is no matching process here.

Formal description

```
supply$_\theta^M$(user, amount):
    update_$\lambda_\theta^M$()
    transfer$_\theta$(user, $\overline{M_\psi}$, amount)

    // Peer-to-peer supply

    //// Match peer-to-peer borrow delta
    matched = min($\delta_\theta^B$ * $\lambda_\theta^B$, amount)
    $\delta_\theta^B$ -= matched / $\lambda_\theta^B$
    $s_\theta^{P2P}$(user) += matched / $\lambda_\theta^{\gamma_S}$
    $S^{P2P}$ += matched / $\lambda_\theta^{\gamma_S}$
    amount -= matched
    repay$_\theta$($\overline{M_\psi}$, matched)

    //// Promote borrowers
    (matched, _) = matchBorrowers$_\theta$(amount, maxGasForMatching)
    $s_\theta^{P2P}$(user) += matched / $\lambda_\theta^{\gamma_S}$
    $S^{P2P}$ += matched / $\lambda_\theta^{\gamma_S}$
    $B^{P2P}$ += matched / $\lambda_\theta^{\gamma_B}$
    amount -= matched
    repay$_\theta$($\overline{M_\psi}$, matched)

    // Pool supply

    $s_\theta^{Pool}$(user) += amount / $\lambda_\theta^S$
    supply$_\theta$($\overline{M_\psi}$, amount)
```

6.2.2 Borrow

High-level description

1. **Peer-to-peer borrow:**
   (a) **Match peer-to-peer supply delta:** Morpho "matches" the user with the peer-to-peer supply delta and reduces it. This is done first to keep the delta as low as possible. A `withdraw` on the pool is performed to reduce the surplus of supply on the pool induced by the peer-to-peer supply delta. Note that this allows responsive users to skip the queue of users on the pool and be instantly matched peer-to-peer.

   (b) **Promote suppliers:** Morpho matches the incoming demand with some supply on the pool. The matching engine tries to match as much liquidity as possible from pool suppliers. A `withdraw` on the pool is performed to disconnect the matched suppliers from it.

2. **Pool borrow:**
   (a) **No matching process:** The remaining demand is met on the underlying pool with a call to `borrow`.

Formal description

```
borrow$_\theta^M$(user, amount):
    update_$\lambda_\theta^M$()
```

```
if amount * $p_\theta$ > borrowing_capacity$^M$(a):
    return
initialAmount = amount

// Peer-to-peer borrow

//// Match peer-to-peer supply delta
matched = min($\delta_\theta^S$ * $\lambda_\theta^S$, amount)
$\delta_\theta^S$ -= matched / $\lambda_\theta^S$
$b_\theta^{P2P}$(user) += matched / $\lambda_\theta^{\gamma_B}$
$B^{P2P}$ += matched / $\lambda_\theta^{\gamma_B}$
amount -= matched
withdraw$_\theta$($\overline{M_\psi}$, matched)

//// Promote suppliers
(matched, _) = matchSuppliers$_\theta$(amount, maxGasForMatching)
$b_\theta^{P2P}$(user) += matched / $\lambda_\theta^{\gamma_B}$
$S^{P2P}$ += matched / $\lambda_\theta^{\gamma_S}$
$B^{P2P}$ += matched / $\lambda_\theta^{\gamma_B}$
amount -= matched
withdraw$_\theta$($\overline{M_\psi}$, matched)

// Pool borrow

$b_\theta^{Pool}$(user) += amount / $\lambda_\theta^B$
borrow$_\theta$($\overline{M_\psi}$, amount)

transfer$_\theta$($\overline{M_\psi}$, user, initialAmount)
```

6.2.3 Withdraw

High-level description

1. **Pool withdraw:**

   (a) **No matching process:** If some of the user's liquidity is supplied on the pool, Morpho withdraws the corresponding part of its position on the underlying pool. It favors this option to maximize the user's matched peer-to-peer liquidity and, thus, their capital efficiency. A `withdraw` on the underlying pool is performed.

(b) **Reduce peer-to-peer supply delta:** Morpho reduces the peer-to-peer supply delta if there is one. It favors this option before the transfer withdraw to minimize the delta. A `withdraw` on the pool is performed to reduce the peer-to-peer supply delta.

2. **Peer-to-peer withdraw:**

   (a) **Transfer withdraw:**

      i. **Promote suppliers:** If the user is matched peer-to-peer, Morpho replaces him with other pool suppliers. It favors this option before the breaking withdraw to maximize the total peer-to-peer liquidity. The matching engine tries to match as much liquidity as possible from pool suppliers. The liquidity is retrieved by a call to `withdraw` on the underlying pool.

   (b) **Breaking withdraw:**

      i. **Demote borrowers:** Morpho breaks the peer-to-peer credit lines of the withdrawing user with the matched borrowers and reconnects them to the pool. The matching engine tries to unmatch as much debt as possible from peer-to-peer borrowers. A `borrow` on the pool is performed to reconnect the borrowers to the pool. The borrowed assets are transferred to the withdrawer.

      ii. **Increase peer-to-peer borrow delta:** Suppose the gas available for matching was not enough to free enough liquidity. In that case, Morpho still performs the breaking withdraw, reconnecting the peer-to-peer borrowers to the pool via a borrow on the pool, and increases the peer-to-peer borrow delta. A part of the peer-to-peer matched borrow is now on the pool, and the indexes will be updated accordingly from now, growing at a higher rate.

Formal description

$\texttt{withdraw}^{M}_{\theta}\texttt{(user, amount):}$

```
update_$\lambda_\theta^M$()
if amount * $F_\theta$ * $p_\theta$ > borrowing_capacity$^M$(user):
    return
initialAmount = amount

// Pool withdraw

withdrawnFromPool = min(amount, $s_\theta^{Pool}$(user) * $\lambda_\theta^S$)
$s_\theta^{Pool}$(user) -= withdrawnFromPool / $\lambda_\theta^S$
amount -= withdrawnFromPool
withdraw$_\theta$($\overline{M_\psi}$, withdrawnFromPool)

// Reduce peer-to-peer supply delta
matched = min($\delta_\theta^S$ * $\lambda_\theta^S$, amount)
$\delta_\theta^S$ -= matched / $\lambda_\theta^S$
$s_\theta^{P2P}$(user) -= matched / $\lambda_\theta^{\gamma_S}$
$S^{P2P}$ -= matched / $\lambda_\theta^{\gamma_S}$
amount -= matched
withdraw$_\theta$($\overline{M_\psi}$, matched)

// Peer-to-peer withdraw

//// Transfer withdraw

// Promote suppliers
(matched, gasLeft) = matchSuppliers$_\theta$(amount, maxGasForMatching)
$s_\theta^{P2P}$(user) -= matched / $\lambda_\theta^{\gamma_S}$
amount -= matched
withdraw$_\theta$($\overline{M_\psi}$, matched)

//// Breaking withdraw

// Demote borrowers
(unmatched, _) = unmatchBorrowers$_\theta$(amount, gasLeft)
// Increase peer-to-peer borrow delta
if amount - unmatched > 0:
    $\delta_\theta^B$ += (amount - unmatched) / $\lambda_\theta^B$
```

```
$S^{P2P}$ -= amount / $\lambda_\theta^{\gamma_S}$
$B^{P2P}$ -= unmatched / $\lambda_\theta^{\gamma_B}$
$s_\theta^{P2P}$(user) -= amount / $\lambda_\theta^{\gamma_S}$
borrow$_\theta$($\overline{M_\psi}$, amount)

transfer$_\theta$($\overline{M_\psi}$, user, initialAmount)
```

Notice that the `gasLeft` value returned from the call to `matchSuppliers` determines how much gas the call to `unmatchBorrowers` is allowed to consume.

6.2.4 Repay

High-level description

1. **Pool repay:**

   (a) **No matching process:** If some of the user's debt is borrowed from the pool, Morpho repays liquidity from the underlying pool. It favors this option to maximize the user's matched liquidity and capital efficiency. A `repay` is performed on the underlying pool.

   (b) **Reduce peer-to-peer borrow delta** Morpho reduces the peer-to-peer borrow delta if there is one. It favors this option before the transfer repay to minimize the delta. A `repay` on the pool is performed to reduce the peer-to-peer borrow delta.

2. **Fee repay:** The fee charged by Morpho is accounted when users repay their peer-to-peer positions. The total amount of the fee accumulated over all the users can be computed by subtracting the real amount (subtracting the delta) of supply in peer-to-peer to the real amount of borrow in peer-to-peer. This difference comes from the reserve factor which is making the peer-to-peer borrow index grow at a higher speed than the peer-to-peer-supply index. The peer-to-peer position of the user is updated accordingly to the fee that is repaid, making this operation seamless when interacting with the protocol.

3. **Peer-to-peer repay:**

   (a) **Transfer repay:**

   i. **Promote borrowers:** If the user is matched peer-to-peer, Morpho replaces the peer-to-peer credit lines with other pool borrowers. It favors this option before the breaking repay to maximize the total peer-to-peer liquidity. The matching engine tries to match as much debt as possible from pool borrowers. A `repay` on the pool is performed to cut those borrowers debt on the pool.

(b) **Breaking repay:**

   i. **Demote suppliers:** Morpho breaks the peer-to-peer credit lines of the repaying user with the remaining matched suppliers and deposits the unmatched liquidity on the underlying pool. The matching engine tries to unmatch as much liquidity as possible from peer-to-peer suppliers. Morpho `supply` on the underlying pool, to reconnect the suppliers to the pool.

   ii. **Increase peer-to-peer supply delta:** If the gas available for matching was not enough to free enough liquidity, Morpho still performs the breaking repay (reconnection of peer-to-peer suppliers by a supply on the pool) and increases the peer-to-peer supply delta. A part of the peer-to-peer matched supply is now on the pool, and the indexes will be updated accordingly from now, growing at a lower rate.

Formal description

```
repay$^M_\theta$(user, amount):
    update_$\lambda^M_\theta$()
    if amount > $b^{P2P}_\theta$(user) * $\lambda^{\gamma^B}$ + $b^{Pool}$(user) * $\lambda^B$:
        return
    transfer$_\theta$(user, $\overline{M_\psi}$, amount)

    // Pool repay

    repaidOnPool = min(amount, $b^{Pool}_\theta$(user) * $\lambda^B_\theta$)
    $b^{Pool}_\theta$(user) -= repaidOnPool / $\lambda^B_\theta$
    amount -= repaidOnPool
    repay$_\theta$($\overline{M_\psi}$, repaidOnPool)
```

```
// Reduce peer-to-peer borrow delta
matched = min($\delta_\theta^B$ * $\lambda_\theta^B$, amount)
$\delta_\theta^B$ -= matched / $\lambda_\theta^B$
$b_\theta^{P2P}$(user) -= matched / $\lambda_\theta^{\gamma_B}$
$B^{P2P}$ -= matched / $\lambda_\theta^{\gamma_B}$
amount -= matched
repay$_\theta$($\overline{M_\psi}$, matched)

// Fee repay
totalFee = $(B^{P2P} * \lambda_\theta^{\gamma_B} - \delta_\theta^B * \lambda_\theta^B) - (S^{P2P} * \lambda_\theta^{\gamma_S} - \delta_\theta^S * \lambda_\theta^S)$
feeRepaid = min(totalFee, amount)
$b_\theta^{P2P}$(user) -= feeRepaid / $\lambda_\theta^{\gamma_B}$
$B^{P2P}$ -= feeRepaid / $\lambda_\theta^{\gamma_B}$
amount -= feeRepaid

// Peer-to-peer repay

//// Transfer repay

// Promote borrowers
(matched, gasLeft) = matchBorrowers$_\theta$(amount, maxGasForMatching)
$b_\theta^{P2P}$(user) -= matched / $\lambda_\theta^{\gamma_B}$
amount -= matched
repay$_\theta$($\overline{M_\psi}$, matched)

//// Breaking repay

// Demote suppliers
(unmatched, _) = unmatchSuppliers$_\theta$(amount, gasLeft)
// Increase peer-to-peer supply delta
if amount - unmatched > 0:
    $\delta_\theta^S$ += (amount - unmatched) / $\lambda_\theta^S$
$S^{P2P}$ -= unmatched / $\lambda_\theta^{\gamma_S}$
$B^{P2P}$ -= amount / $\lambda_\theta^{\gamma_B}$
$b_\theta^{P2P}$(user) -= amount / $\lambda_\theta^{\gamma_B}$
supply$_\theta$($\overline{M_\psi}$, amount)
```

6.3 Impact on rates

Assume that λ^{γ^S} and λ^{γ^B} are the peer-to-peer supply index and peer-to-peer borrow index. This notation is introduced to take into account the delta mechanism (in addition to the reserve factor and the peer-to-peer cursor). We want to compute those indexes. We define the supply and borrow share of the delta as follows:

$$\gamma^S = \frac{\delta^S \cdot \lambda^S}{S^{P2P} \cdot \lambda^{\gamma^S}}$$
$$\gamma^B = \frac{\delta^B \cdot \lambda^B}{B^{P2P} \cdot \lambda^{\gamma^B}}$$

We want the part of the matched liquidity corresponding to the delta to grow at a speed that depends on the pool rate (r^S and r^B). The rest should to grow at a speed that depends on the peer-to-peer rate (r^{ρ^S} and r^{ρ^B}). It implies that the resulting rates are, for the supply indexes:

$$\begin{aligned} r^{\gamma^S} &= (1-\gamma^S)r^{\rho^S} + \gamma^S r^S \\ &= (1-\gamma^S)((1-\alpha+\rho\alpha)r^S + (\alpha-\rho\alpha)r^B) + \rho r^S \\ &= (1-(1-\gamma^S)(1-\rho)\alpha)r^S + (1-\gamma^S)(1-\rho)\alpha r^B \end{aligned}$$

and for the borrow indexes:

$$\begin{aligned} r^{\gamma^B} &= (1-\gamma^B)r^{\rho^B} + \gamma^B r^B \\ &= (1-\gamma^B)((1-\rho)(1-\alpha)r^S + (\rho+\alpha-\rho\alpha)r^B) + \gamma^B r^B \\ &= (1-\gamma^B)(1-\rho)(1-\alpha)r^S + (1-(1-\gamma^B)(1-\rho)(1-\alpha))r^B \end{aligned}$$

Let $t, t' \in \mathbb{N}$ with $t' < t$. Using theorem 5.2.1, we get:

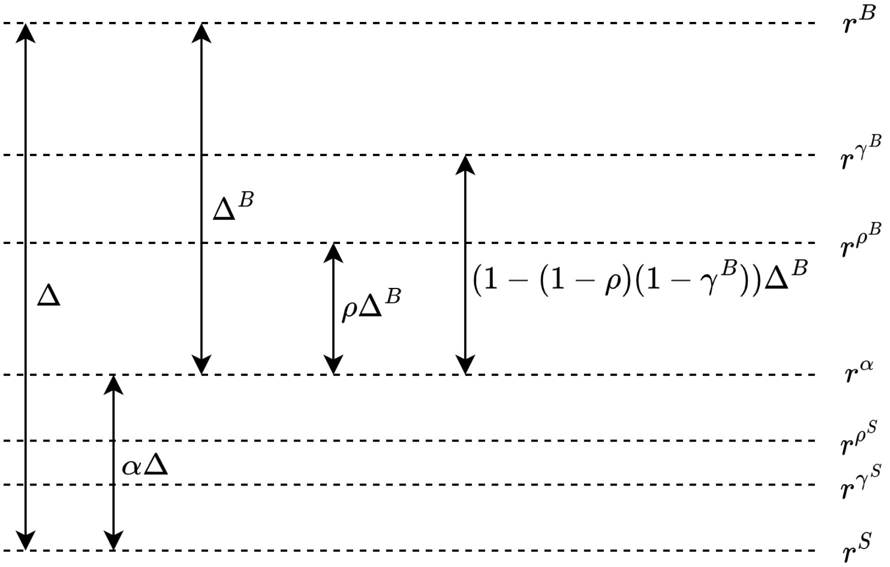
$$\lambda_t^{\gamma^S} \approx \lambda_{t'}^{\gamma^S}[(1-(1-\gamma_{t'}^S)(1-\rho)\alpha)\frac{\lambda_t^S}{\lambda_{t'}^S} + (1-\gamma_{t'}^S)(1-\rho)\alpha\frac{\lambda_t^B}{\lambda_{t'}^B}]$$
$$\lambda_t^{\gamma^B} \approx \lambda_{t'}^{\gamma^B}[(1-\gamma_{t'}^B)(1-\rho)(1-\alpha)\frac{\lambda_t^S}{\lambda_{t'}^S} + (1-(1-\gamma_{t'}^B)(1-\rho)(1-\alpha))\frac{\lambda_t^B}{\lambda_{t'}^B}]$$

Those formulas are efficient computations that come from approximating the desired rates when taking into account the reserve factor and the delta mechanism. The Morpho protocol does not update the rates but updates the indexes instead, using the formulas above.

$$\lambda_t^{\gamma^S} = \lambda_{t'}^{\gamma^S}[(1-(1-\gamma_{t'}^S)(1-\rho)\alpha)\frac{\lambda_t^S}{\lambda_{t'}^S} + (1-\gamma_{t'}^S)(1-\rho)\alpha\frac{\lambda_t^B}{\lambda_{t'}^B}] \tag{9}$$

$$\lambda_t^{\gamma^B} = \lambda_{t'}^{\gamma^B}[(1-\gamma_{t'}^B)(1-\rho)(1-\alpha)\frac{\lambda_t^S}{\lambda_{t'}^S} + (1-(1-\gamma_{t'}^B)(1-\rho)(1-\alpha))\frac{\lambda_t^B}{\lambda_{t'}^B}] \tag{10}$$

To sum up, the pool's spread is $\Delta = r^B - r^S$ and two factors can increase Morpho's peer-to-peer spread: the reserve factor and the deltas. The difference $r^B - r^\alpha$ is denoted by Δ^B in the following diagram and the reserve factor ρ can be seen as a proportion of the improvement that is taken on the borrow and supply side, without taking into account the deltas. Similarly, the peer-to-peer deltas can be seen as the proportion of the peer-to-peer amount that is actually on the pool. We see that those two factors are multiplicative: the proportion of the improvement that is lost on the rate r^{γ^B} is given by $1-(1-\rho)(1-\gamma^B)$. We can thus see the reserve factor as a cut on the improvement compared to the pool, taking into account the deltas.



The following theorem gives us an intuition about how the deltas evolve if they are not reduced by new interactions with the protocol.

Theorem 6.3.1. *Between two updates with no other interaction on Morpho, the share of the supply delta γ^S is decreasing, and the share of the borrow delta γ^B is increasing.*

Proof. We make use of the previous notations and formulas 9 and 10. Since we are looking at the delta shares between two updates with no interaction with Morpho in between, the quantities δ^S, δ^B, S^{P2P}, and B^{P2P} are constant. Thus, we only have to show that:

$$\frac{\lambda^S_{t'}}{\lambda^{\gamma^S}_{t'}} \cdot \frac{\lambda^{\gamma^S}_t}{\lambda^S_t} \geq 1$$

$$\frac{\lambda^B_{t'}}{\lambda^{\gamma^B}_{t'}} \cdot \frac{\lambda^{\gamma^B}_t}{\lambda^B_t} \leq 1$$

We have

$$\begin{aligned}\frac{\lambda^S_{t'}}{\lambda^{\gamma^S}_{t'}} \cdot \frac{\lambda^{\gamma^S}_t}{\lambda^S_t} &= \frac{\lambda^S_{t'}}{\lambda^S_t} \cdot [(1-\gamma^B_{t'})(1-\rho)(1-\alpha)\frac{\lambda^S_t}{\lambda^S_{t'}} + (1-(1-\gamma^B_{t'})(1-\rho)(1-\alpha))\frac{\lambda^B_t}{\lambda^B_{t'}}] \\ &= (1-\gamma^B_{t'})(1-\rho)(1-\alpha) + (1-(1-\gamma^B_{t'})(1-\rho)(1-\alpha)) \cdot \frac{\lambda^B_t}{\lambda^B_{t'}} \cdot \frac{\lambda^S_{t'}}{\lambda^S_t}\end{aligned}$$

Since the borrow index grows more rapidly on the pool than the supply index, we have $\frac{\lambda^B_t}{\lambda^B_{t'}} \cdot \frac{\lambda^S_{t'}}{\lambda^S_t} \geq 1$. This implies that

$$\frac{\lambda^S_{t'}}{\lambda^{\gamma^S}_{t'}} \cdot \frac{\lambda^{\gamma^S}_t}{\lambda^S_t} \leq (1-\gamma^B_{t'})(1-\rho)(1-\alpha) + (1-(1-\gamma^B_{t'})(1-\rho)(1-\alpha)) = 1$$

■

6.4 Non-liquidation theorem

With the addition of the delta mechanism, the lemmas 3.4.1 and 5.3.2 no longer hold. We redefine them to prove the non-liquidation theorem again.

Lemma 6.4.1. *(Pool position's integrity) Morpho's position on the pool is always the sum of the positions on the pool of all its users, plus the deltas.*

$$\forall \theta \in \Theta_{M_\psi}, s_\theta(\overline{M_\psi}) = \sum_u s^{Pool}_\theta(u) + \delta^S_\theta$$

$$\forall \theta \in \Theta_{M_\psi}, b_\theta(\overline{M_\psi}) = \sum_u b^{Pool}_\theta(u) + \delta^B_\theta$$

Proof. A proof similar to the one of the lemma 3.4.1 applies here also. ■

Lemma 6.4.2. *(Peer-to-peer inequality) The sum of the suppliers' peer-to-peer balances minus the peer-to-peer supply delta is always less than or equal to the sum of the borrowers' peer-to-peer balances minus the peer-to-peer borrow delta.*

$$\forall \theta \in \Theta_{M_\psi}, \sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_S} - \delta_\theta^S \cdot \lambda_\theta^S \leq \sum_u b_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_B} - \delta_\theta^B \cdot \lambda_\theta^B$$

Proof. We can do the same kind of proof as for 3.4.2.

Base case The inequality is clearly verified at deployment.

Induction step Let $\sigma_i \in \Sigma$ be a state where the inequality is verified. Let a transition which transits to the state $\sigma_r \in \Sigma$.

We treat the case where the transition is associated to the function call to `withdraw`. By now, we only consider the market θ, as others are untouched, so theta can be omitted as a subscript. Let $(\theta, u, x) \in D_{\texttt{withdraw}}$ be the input for this function call.

We first consider the update of the indexes and try to prove that the equality still holds at the state $\sigma_{r'}$, an intermediary state just after the indexes update. Remark that the variables δ^S and S^{P2P} do not change when updating the index. This means that we have $\sigma_{r'}(\delta^S) = \sigma_i(\delta^S)$ and $\sigma_{r'}(S^{P2P}) = \sigma_i(S^{P2P})$.

We note $\sigma_i(\gamma^S)$ for:

$$\frac{\sigma_i(\delta^S) \cdot \sigma_i(\lambda^S)}{\sigma_i(S^{P2P}) \cdot \sigma_i(\lambda^{\gamma^S})}$$

Let us first consider this relation:

$$\begin{aligned}
L &= \sum_u \sigma_{r'}(s_\theta^{P2P}(u)) \cdot \sigma_{r'}(\lambda_\theta^{\gamma_S}) - \sigma_{r'}(\delta_\theta^S) \cdot \sigma_{r'}(\lambda_\theta^S) \\
&= \sum_u \sigma_i(s_\theta^{P2P}(u)) \cdot \sigma_i(\lambda_\theta^{\gamma_S}) \cdot \frac{\sigma_{r'}(\lambda_\theta^{\gamma_S})}{\sigma_i(\lambda_\theta^{\gamma_S})} - \sigma_i(\delta_\theta^S) \cdot \sigma_i(\lambda_\theta^S) \cdot \frac{\sigma_{r'}(\lambda_\theta^S)}{\sigma_i(\lambda_\theta^S)} \\
&= \sum_u \sigma_i(s_\theta^{P2P}(u)) \cdot \sigma_i(\lambda_\theta^{\gamma_S}) \cdot \left(\frac{\sigma_{r'}(\lambda_\theta^{\gamma_S})}{\sigma_i(\lambda_\theta^{\gamma_S})} - \sigma_i(\gamma_\theta^S) \cdot \frac{\sigma_{r'}(\lambda_\theta^S)}{\sigma_i(\lambda_\theta^S)} \right)
\end{aligned}$$

Translating the formula 7 into this state setting, we get:

$$\frac{\sigma_{r'}(\lambda_\theta^{\rho^S})}{\sigma_i(\lambda_\theta^{\rho^S})} = (1-(1-\rho_\theta^S)\alpha)\cdot\frac{\sigma_{r'}(\lambda_\theta^S)}{\sigma_i(\lambda_\theta^S)} + \\ (1-\rho_\theta^S)\alpha\cdot\frac{\sigma_{r'}(\lambda_\theta^B)}{\sigma_i(\lambda_\theta^B)}$$

Similarly, for 9, we get:

$$\frac{\sigma_{r'}(\lambda_\theta^{\gamma^S})}{\sigma_i(\lambda_\theta^{\gamma^S})} = (1-(1-\sigma_i(\gamma_\theta^S))(1-\rho_\theta^S)\alpha)\cdot\frac{\sigma_{r'}(\lambda_\theta^S)}{\sigma_i(\lambda_\theta^S)} + \\ (1-\sigma_i(\gamma_\theta^S))(1-\rho_\theta^S)\alpha\cdot\frac{\sigma_{r'}(\lambda_\theta^B)}{\sigma_i(\lambda_\theta^B)}$$

so:

$$\frac{\sigma_{r'}(\lambda_\theta^{\gamma_S})}{\sigma_i(\lambda_\theta^{\gamma_S})} = (1-\sigma_i(\gamma_\theta^S))\cdot\frac{\sigma_{r'}(\lambda_\theta^{\rho_S})}{\sigma_i(\lambda_\theta^{\rho_S})} + \sigma_i(\gamma_\theta^S)\cdot\frac{\sigma_{r'}(\lambda_\theta^S)}{\sigma_i(\lambda_\theta^S)}$$

Using this last equality, we get:

$$\begin{aligned} L &= \sum_u \sigma_i(s_\theta^{P2P}(u))\cdot\sigma_i(\lambda_\theta^{\gamma_S})\cdot(1-\sigma_i(\gamma_\theta^S))\cdot\frac{\sigma_{r'}(\lambda_\theta^{\rho_S})}{\sigma_i(\lambda_\theta^{\rho_S})} \\ &= \frac{\sigma_{r'}(\lambda_\theta^{\rho_S})}{\sigma_i(\lambda_\theta^{\rho_S})}\left(\sum_u \sigma_i(s_\theta^{P2P}(u))\cdot\sigma_i(\lambda_\theta^{\gamma_S}) - \sigma_i(\delta_\theta^S)\cdot\sigma_i(\lambda_\theta^S)\right) \end{aligned}$$

Similarly:

$$\begin{aligned} R &= \sum_u \sigma_{r'}(b_\theta^{P2P}(u))\cdot\sigma_{r'}(\lambda_\theta^{\gamma_B}) - \sigma_{r'}(\delta_\theta^B)\cdot\sigma_{r'}(\lambda_\theta^B) \\ &= \frac{\sigma_{r'}(\lambda_\theta^{\rho_B})}{\sigma_i(\lambda_\theta^{\rho_B})}\left(\sum_u \sigma_i(b_\theta^{P2P}(u))\cdot\sigma_i(\lambda_\theta^{\gamma_B}) - \sigma_i(\delta_\theta^B)\cdot\sigma_i(\lambda_\theta^B)\right) \end{aligned}$$

Additionally, we notice that the growth of the peer-to-peer supply index without taking into account the deltas is smaller than the growth of the peer-to-peer borrow index:

$$\frac{\sigma_{r'}(\lambda^{\rho_S})}{\sigma_i(\lambda^{\rho_S})} \leq \frac{\sigma_{r'}(\lambda^{\rho_B})}{\sigma_i(\lambda^{\rho_B})}$$

So, because the inequality is verified at the state σ_i:

$$\begin{aligned}\sum_u \sigma_{r'}(s_\theta^{P2P}(u)) \cdot \sigma_{r'}(\lambda_\theta^{\gamma_S}) - \sigma_{r'}(\delta_\theta^S) \cdot \sigma_{r'}(\lambda_\theta^S)&\\ \leq \sum_u \sigma_{r'}(b_\theta^{P2P}(u)) \cdot \sigma_{r'}(\lambda_\theta^{\gamma_B}) - \sigma_{r'}(\delta_\theta^B) \cdot \sigma_{r'}(\lambda_\theta^B)&\end{aligned}$$

We proved that the inequality holds at $\sigma_{r'}$ if it held at σ_i. Now we will focus on σ_r, the state at the end of the transition. From now, we will write λ^S for $\sigma_r(\lambda_\theta^S) = \sigma_{r'}(\lambda_\theta^S)$, and we do the same for $\sigma_r(\lambda_\theta^B)$, $\sigma_r(\lambda_\theta^{\gamma_S})$ and $\sigma_r(\lambda_\theta^{\gamma_B})$.

We do not go through the **pool withdraw** step, as it does not change the peer-to-peer balance of any user nor any delta. Let us consider the **reduce peer-to-peer supply delta** step. Let x^{δ_S} be the amount that is reduced from the delta in the underlying unit (the first `matched` in the pseudo-code). The peer-to-peer supply delta is reduced by x^{δ_S}/λ^S, and the peer-to-peer supply balance of the user is reduced by $x^{\delta_S}/\lambda^{\gamma_S}$.

Consider also the **promote suppliers** step. Let x^{μ_S} be the total amount matched in underlying by `matchSuppliers` (the second `matched` in the pseudo-code). The peer-to-peer supply balance of the user is reduced by $x^{\mu_S}/\lambda^{\gamma_S}$. And the sum of the other suppliers' balances is increased by $x^{\mu_S}/\lambda^{\gamma_S}$.

Finally, consider the **demote borrowers** step. Let x^{μ_B} be the total amount unmatched in underlying by `unmatchBorrowers` (`unmatched` in the pseudo-code). Let x^{δ_B} be the amount that is added to the borrow delta in the underlying unit (`amount - unmatched` in the pseudo-code). The peer-to-peer supply balance of the user is reduced by $(x^{\mu_B}+x^{\delta_B})/\lambda^{\gamma_S}$, and the sum of the peer-to-peer borrow balances is reduced by $x^{\mu_B}/\lambda^{\gamma_B}$. The peer-to-peer borrow delta is increased by x^{δ_B}/λ^B. Finally:

$$\begin{aligned}
L =& \sum_u \sigma_r(s^{P2P}(u)) \cdot \lambda^{\gamma_S} - \sigma_r(\delta^S) \cdot \lambda^S \\
=& \left(\sum_u \sigma_{r'}(s^{P2P}(u)) - \frac{x^{\delta_S}}{\lambda^{\gamma_S}} - \frac{x^{\mu_S}}{\lambda^{\gamma_S}} + \frac{x^{\mu_S}}{\lambda^{\gamma_S}} - \frac{(x^{\mu_B} + x^{\delta_B})}{\lambda^{\gamma_S}} \right) \cdot \lambda^{\gamma_S} - \\
& \left(\sigma_{r'}(\delta^S) - \frac{x^{\delta_S}}{\lambda^S} \right) \cdot \lambda^S \\
=& \sum_u \sigma_{r'}(s^{P2P}(u)) \cdot \lambda^{\gamma_S} - \sigma_{r'}(\delta^S) \cdot \lambda^S - x^{\mu_B} - x^{\delta_B} \\
R =& \sum_u \sigma_r(b^{P2P}(u)) \cdot \lambda^{\gamma_B} - \sigma_r(\delta^B) \cdot \lambda^B \\
=& \left(\sum_u \sigma_{r'}(b^{P2P}(u)) - \frac{x^{\mu_B}}{\lambda^{\gamma_B}} \right) \cdot \lambda^{\gamma_B} - \left(\sigma_{r'}(\delta^B) + \frac{x^{\delta_B}}{\lambda^B} \right) \cdot \lambda^B \\
=& \sum_u \sigma_{r'}(b^{P2P}(u)) \cdot \lambda^{\gamma_B} - \sigma_{r'}(\delta^B) \cdot \lambda^B - x^{\mu_B} - x^{\delta_B}
\end{aligned}$$

So:

$$\begin{aligned}
& \sum_u \sigma_r(s^{P2P}(u)) \cdot \lambda^{\gamma_S} - \sigma_r(\delta^S) \cdot \sigma_r(\lambda^S) \\
& \qquad \leq \sum_u \sigma_r(b^{P2P}(u)) \cdot \lambda^{\gamma_B} - \sigma_r(\delta^B) \cdot \sigma_r(\lambda^B)
\end{aligned}$$

We could do the same proofs for all the other external functions and for Γ_{tick}. ■

We come back to the non-liquidation theorem 3.4.1, with an updated proof taking into account the delta mechanism.

Theorem 6.4.1 (Non-liquidation with deltas)**.** *If Morpho is equipped with a working liquidation system, then Morpho's position on pool is not liquidatable, even if in the presence of deltas. Thus the following inequality still stands:*

$$\sum_\theta p_\theta \cdot b_\theta(\overline{M_\psi}) \cdot \lambda_\theta^B \leq \sum_\theta p_\theta \cdot F_\theta \cdot s_\theta(\overline{M_\psi}) \cdot \lambda_\theta^S$$

Proof. The proof is similar to the one without the reserve factor and the delta mechanism. What is changing is mostly the assumptions.

From the peer-to-peer inequality lemma 6.4.2, we know that, for each token θ, the following inequality stands:

$$\sum_u b_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_B} - \delta_\theta^B \cdot \lambda_\theta^B \geq \sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_S} - \delta_\theta^S \cdot \lambda_\theta^S$$

We also get the following equalities from the pool position's integrity lemma 6.4.1:

$$b_\theta(\overline{M_\psi}) = \delta_\theta^B + \sum_u b_\theta^{Pool}(u)$$

$$s_\theta(\overline{M_\psi}) = \delta_\theta^S + \sum_u s_\theta^{Pool}(u)$$

We still have the following inequality from the assumptions of the theorem:

$$\sum_\theta p_\theta \cdot \sum_u \omega_\theta^B(u) \leq \sum_\theta p_\theta \cdot F_\theta \cdot \sum_u \omega_\theta^S(u)$$

Let us call L and R the left-hand side and the right-hand side of the previous inequality, and reduce them. For L:

$$\begin{aligned}
L =& \sum_\theta p_\theta \cdot \sum_u \omega_\theta^B(u) \\
=& \sum_\theta p_\theta \cdot \sum_u (b_\theta^{Pool}(u) \cdot \lambda_\theta^B + b_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma^B}) \\
=& \sum_\theta p_\theta \cdot ((b_\theta(\overline{M_\psi}) - \delta_\theta^B) \cdot \lambda_\theta^B + \sum_u b_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_B}) \\
=& \sum_\theta p_\theta \cdot b_\theta(\overline{M_\psi}) \cdot \lambda_\theta^B + \sum_\theta p_\theta \cdot (\sum_u b_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_B} - \delta_\theta^B \cdot \lambda_\theta^B) \\
\geq& \sum_\theta p_\theta \cdot b_\theta(\overline{M_\psi}) \cdot \lambda_\theta^B + \sum_\theta p_\theta \cdot (\sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_S} - \delta_\theta^S \cdot \lambda_\theta^S)
\end{aligned}$$

And for R:

$$\begin{aligned}
R =& \sum_\theta p_\theta \cdot F_\theta \cdot \sum_u \omega_\theta^S(u) \\
=& \sum_\theta p_\theta \cdot F_\theta \cdot \sum_u (s_\theta^{Pool}(u) \cdot \lambda_\theta^S + s_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_S}) \\
=& \sum_\theta p_\theta \cdot F_\theta \cdot ((s_\theta(\overline{M_\psi}) - \delta_\theta^S) \cdot \lambda_\theta^S + \sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_S}) \\
=& \sum_\theta p_\theta \cdot F_\theta \cdot s_\theta(\overline{M_\psi}) \cdot \lambda_\theta^S + \sum_\theta p_\theta \cdot F_\theta \cdot (\sum_u s_\theta^{P2P}(u) \cdot \lambda_\theta^{\gamma_S} - \delta_\theta^S \cdot \lambda_\theta^S)
\end{aligned}$$

So, because $F_\theta \leq 1$, we can still conclude that:

$$\sum_\theta p_\theta \cdot b_\theta(\overline{M_\psi}) \cdot \lambda_\theta^B \leq \sum_\theta p_\theta \cdot F_\theta \cdot s_\theta(\overline{M_\psi}) \cdot \lambda_\theta^S$$

■

7 Conclusion

The Morpho protocol is an improvement of the established protocols for loanable funds (see section 2) that have seen a significant rise in popularity in the past years. Such protocols have numerous advantages, including high liquidity with the possibility to withdraw or borrow significant amounts at any time. They also define robust risk parameters which minimize the risk of insolvencies and liquidations in high-volatility markets but require notably leaving a fraction of the liquidity idle on the contract. The main idea of Morpho, explained in section 3, is to improve capital efficiency by matching peer-to-peer the incoming liquidity as much as possible while ensuring the same liquidity and at least the same rates as the underlying lending pool by aggregating the remaining positions on it. In doing so, we are faced with different challenges. First, we should ensure that the protocol enjoys the same liquidity as the underlying lending pool. It amounts to finding a fallback mechanism, notably including the possibility to withdraw when matched peer-to-peer. A solution to this problem was given in section 3.3, with detailed steps explaining the process. Second, we should make sure that the position taken on the pool is safe. This point has been formalized as the *non-liquidation theorem*, and a clear proof was given in section 3.4 and refined later in section 6.4. Third, we need relevant rates and a robust system to account for them. In section 5, we turn to a general analysis of it, and we prove that Morpho users enjoy improved rates compared to users of the underlying pool. Finally, one last challenge is to ensure that the protocol can scale by bounding the gas usage of any transaction, no matter its size. This problem comes from the matching engine, described in section 4, as it has to scale with the number of users. Section 6 returns to the scalability problem with a solution called the delta mechanism, where proofs and previous results are updated to consider this new mechanism.

References

[88m22] 88mph. 88mph Documentation, 2022. Available at https://docs.88mph.app.

[Aav22] Aave. Aave Documentation, 2022. Available at https://docs.aave.com/hub.

[APW22] APWine. APWine Documentation, 2022. Available at https://docs.apwine.fi/.

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. 2014. Available at https://ethereum.org/en/whitepaper.

[Com22] Compound. Compound Documentation, 2022. Available at https://docs.compound.finance.

[DFB+21] Mathis Gontier Delaunay, Paul Frambot, Katia Babbar, Vincent Danos, and Adrien Husson. Morpho: Optimizing decentralized liquidity protocols. 2021. Available at https://whitepaper.morpho.xyz.

[GWPK20] Lewis Gudgeon, Sam Werner, Daniel Perez, and William J Knottenbelt. Defi protocols for loanable funds: Interest rates, liquidity and market efficiency. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020.

[Not22] Notional. Notional Whitepaper, 2022. Available at https://docs.notional.finance/developers/whitepaper/whitepaper.

[PWXL20] Daniel Perez, Sam M Werner, Jiahua Xu, and Benjamin Livshits. Liquidations: Defi on a knife-edge. *arXiv preprint arXiv:2009.13235*, 2020.

[RN20] Dan Robinson and Allan Niemerg. The Yield Protocol, On-Chain Lending With Interest Rate Discovery. 2020. Available at https://yield.is/Yield.pdf.

[SEN17] Ari Juels Steve Ellis and Sergey Nazarov. ChainLink, A Decentralized Oracle Network. September 2017. Available at https://research.chain.link/whitepaper-v1.pdf.

[Sen22] Sense. Sense Documentation, 2022. Available at https://docs.sense.finance/docs/faq.

[sol22] Solidity Documentation, 2022. Available at https://docs.soliditylang.org/en/latest.

[Tea] Uniswap Team. Uniswap Oracle Documentation. Available at https://docs.uniswap.org/contracts/v2/concepts/core-concepts/oracles.

[VB15] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard, 2015. Available at https://eips.ethereum.org/EIPS/eip-20.