Paradigm
(https://www.paradigm.xyz)

TWAMM

7.28.2021 | Dave White (https://www.paradigm.xyz/team/davewhite/), Dan Robinson (https://www.paradigm.xyz/team/danrobinson/), Hayden Adams (https://twitter.com/haydenzadams)

Contents

Introduction

This paper introduces a new type of automated market maker, or AMM, that helps traders on Ethereum efficiently execute large orders.

We call it the *time-weighted average market maker*, or TWAMM (pronounced “tee-wham”).

It works by breaking long-term orders into infinitely many infinitely small pieces and executing them against an embedded constant-product AMM smoothly over time.

Summary

Let’s say Alice wants to buy 100 million USDC worth of ETH on-chain. An order of this size will be expensive to execute on existing AMMs like Uniswap, which must charge Alice a high price in case she knows something they don’t.

Today, Alice’s best option is to manually break her order up into several pieces and execute them over a period of hours, giving the market time to realize she does not have inside information so it can give her a better price.

If she sends a few large sub-orders, each will still have significant price impact and will be vulnerable to *sandwich attacks* by adversarial traders. On the other hand, if she sends many small sub-orders, she will have to take on all the work and risk of active trading, and will pay high costs in the form of per-transaction gas fees to miners.

The TWAMM resolves this dilemma for Alice by trading on her behalf. It breaks her order into infinitely many infinitely small *virtual orders* to ensure perfectly smooth execution over time, and, using a special mathematical relationship with its embedded AMM, is able to amortize gas costs across these virtual orders. Because it processes trades *in between* blocks, it is also less susceptible to sandwich attacks.

Market Making Basics

Market Makers

Consider a market for two financial assets, say, USDC and ETH. A *market maker* is a participant in this market who is willing, at any time, to trade either one for the other.

If you have 100 million USDC and want to use it to buy ETH, you probably won't be able to find another individual who wants to do the opposite trade at the exact same time. Instead, you will most likely go to a market populated by one or more market makers and trade with them.

Adverse Selection

Market makers earn a profit from the *spread*, effectively a fee they charge on each trade. They lose money when prices *move against* them — when they buy an asset whose price then goes down, or sell an asset whose price then goes up.

Unfortunately for market makers, prices tend to move against them on average. This phenomenon is known as *adverse selection.* It happens because traders with information about future price movements are more likely to trade against market makers for large size.

The most dangerous orders are ones that are both *large* and *urgent*, because those are precisely the type of orders informed traders tend to make. As a result, the most basic market making tactic is to *fade* incoming orders, adjusting prices up when a large buy order comes in, and adjusting prices down when a large sell order comes in.

Automated Market Makers

In the past year, *automated market makers (AMMs)*, led by Uniswap (https://info.uniswap.org/#/), have become hugely popular on Ethereum, handling billions of dollars of volume daily. True to their name, AMMs automate much of the market making process.

The Constant Product Formula

The *constant product formula* is a simple rule that allows anybody to spin up both a new market and a new AMM for a new pair of assets instantaneously.

To create a new Constant Product AMM (CPAMM) between two assets X and Y, a user, called a *liquidity provider*, or LP, deposits *reserves* x and y of those two assets.

The ratio of these assets at any given time represents the *instantaneous price* on the AMM, or the price it will charge for a very small order. For example, if a CPAMM contains 2,000 USDC and 1 ETH in its reserves, its instantaneous price for ETH will be 2,000 USDC.

When traders come to trade with the AMM, it decides what price to give them based on the formula **x * y = k, where x and y are the reserve sizes and k is a constant. This means that the *product* of its reserve sizes stays the same during a trade (ignoring fees).

Example

Consider an ETH/USDC CPAMM with 2,000 USDC and 1 ETH in the reserves, so that x = 2,000, y = 1, and x * y = k = 2,000. The instantaneous price of this AMM is 2,000 / 1 = 2,000 USDC per ETH.

If a trader comes and buys 2,000 USDC worth of ETH, that means they are depositing 2,000 USDC into the X reserves, so that we will have x = 2,000 + 2,000 = 4,000.

Then, since k = 2000, we must have y = x/k = 2000/4000 = 0.5 after the trade. Since y was originally 1, 1 – 0.5 = 0.5 ETH must have gone to the trader.

Since the trader bought 0.5 ETH with their 2000 USDC, they paid an average price of 4,000 USDC per ETH. This high price relative to the instantaneous price reflects the large order size relative to the liquidity in the AMM.

Price Impact and Adverse Selection

In the above case, the trader had to pay 4,000 USDC per ETH for their large order when the cost for a small order would have been only 2,000 USDC per ETH. This difference in price is referred to as the *price impact* of the order. The larger the incoming order, the greater the price impact.

This is how the AMM combats adverse selection: large incoming orders are more likely to be informed, and so the AMM makes them pay a high price. It is the automated equivalent of fading an order.

Executing Large Orders on Current AMMs

Manual Order Splitting

As we can see, executing a large order on an AMM in a single transaction is expensive by design. This excellent article (https://www.paradigm.xyz/2021/04/understanding-automated-market-makers-part-1-price-impact/) goes into the problem in depth and recommends some solutions.

In short, a trader who wishes to execute a large order on an AMM should not do it in a single transaction: they are better off splitting their order into several pieces. This could involve sending orders to several AMMs at once, but these AMMs too will have limited liquidity at any given point in time. The larger the

order, the more attractive splitting it up over *time* becomes.

For example, let's say an investor wants to buy 100 million USDC of ETH on-chain. They don't have any short-term information about the price of ETH, and so don't mind if their order takes some time to execute. In this case, they might break their order into ten chunks of ten million USDC each and execute each chunk one hour apart, limiting the price impact of each part.

The Sub-Order Size Tradeoff

Obviously, if a very large order is split into only a few pieces, each individual sub-order will still be large, and will incur price impact accordingly. Splitting the order into even smaller pieces helps, but this introduces two new problems of its own.

The first problem is *operational complexity*, meaning increased risk and work. The trader might enter the wrong trade quantity for a given trade, or the wrong direction. Or her computer could crash, preventing her from executing part of the order. Even if everything goes right, the process takes time and effort, distracting attention from more profitable endeavors.

The second problem is that each trade incurs fixed *transaction costs*, such as the *gas* paid to Ethereum miners to process the transaction. If the trader splits her order into too many pieces, she can end up spending more money on fees than on actually buying ETH.

Analogy to Traditional Finance

In the world of traditional finance, if an investor or institution wanted to buy $100 million dollars of Apple stock, they would not send a $100 million market buy order directly to an exchange. Nor would they send ten such orders for $10 million each. Breaking the order into pieces much smaller than that would be impractical for most without dedicated trading staff and infrastructure.

Instead, they would most likely send their large order to a broker, who would put on an *algorithmic trade (https://en.wikipedia.org/wiki/Algorithmic_trading)* for them in exchange for a fee. The broker would execute the trade over a specified period of time, say, eight hours, at a price similar to some benchmark. The broker would have a team specialized in executing such trades safely and cheaply.

The TWAP Order

Perhaps the most basic type of algorithmic trade is the Time-Weighted Average Price (https://en.wikipedia.org/wiki/Time-weighted_average_price), or TWAP (pronounced "tee-whap") order. A TWAP order to buy $100 million dollars of Apple stock over eight hours would, as the name suggests, get filled at a price close to the time-weighted average price of Apple stock over that period.

For example, if Apple stock spent four hours priced at $100 and four hours priced at $120 over the time period, the time-weighted average price would be ($100 * 4 + $120 * 4)/8 = $110, and the broker would execute the TWAP order for near that price.

Details vary, but the broker would most likely execute this trade by splitting it into many small pieces throughout the day and sending them to the market. Buying $100 million of Apple stock over eight hours is equivalent to buying around $350 of Apple stock every 100 milliseconds, and we might expect the broker to do more or less that.

The broker has the infrastructure to reduce or eliminate the operational complexity of so many small trades, and, as they have a direct connection to the market, will likely not have to pay much in transaction costs.

The Time-Weighted Average Market Maker

The Time-Weighted Average Market Maker (TWAMM) provides the on-chain equivalent of the TWAP order. The TWAMM has specialized logic for order splitting and a direct connection to an embedded exchange, providing smooth execution for low gas cost. Arbitrageurs keep prices on the TWAMM's embedded exchange in line with market prices, ensuring execution near the time-weighted average price of the asset.

Overview

Each TWAMM instance facilitates trading between a particular pair of assets, such as ETH and USDC.

The TWAMM contains an *embedded AMM*, a standard constant-product market maker for those two assets. Anyone may trade with this embedded AMM at any time, just as if it were a normal AMM.

Traders can submit *long-term orders* to the TWAMM, which are orders to sell a fixed amount of one of the assets over a fixed number of blocks — say, an order to sell 100 ETH over the next 2,000 blocks.

The TWAMM breaks these long-term orders into infinitely many infinitely small *virtual sub-orders*, which trade against the embedded AMM at an even rate over time. Processing transactions for each of these virtual sub-orders individually would cost infinite gas, but a closed-form mathematical formula allows us to calculate their cumulative effect only when needed.

The execution of long-term orders will push the embedded AMM's price away from prices on other markets over time. When this happens, *arbitrageurs* will trade against the embedded AMM's price to bring it back in line, ensuring good execution for long-term orders.

For example, if long-term sells have made ETH cheaper on the embedded AMM than it is on a particular centralized exchange, arbitrageurs will buy ETH from the embedded AMM, bringing its price back up, and sell it on the centralized exchange for a profit.

Ethereum Refresher

Blocks

Ethereum bundles transactions into sequential groups called *blocks*, approximately one every 13 seconds. For the purposes of this post, we will number each block: block 1 is followed by block 2, which is followed by block 3, and so on.

Miners

A distributed groups of *miners* competes to process each block. Anybody with an internet connection can become a miner. This means that programs like AMMs that run on Ethereum cannot keep any secrets: everybody has to be able to calculate exactly what they will do given their inputs.

Gas

Computation on Ethereum is a scarce resource, and so users must pay miners for it in the form of *gas*. The more computation involved in a given transaction, the more gas it will consume. This gas cost is paid entirely by the person submitting the transaction.

Basic Design

Long-Term Orders

Alice wants to buy 100 million USDC worth of ETH over the next 8 hours, or about 2,000 blocks. She enters a long-term order into the TWAMM to buy 100 million USDC worth of ETH over 2,000 blocks, or 50,000 USDC per block.

As mentioned above, we don't know in advance which miners will process future transactions on the TWAMM. This means Alice's order must be visible to everyone, introducing information leakage concerns which we discuss below.

Order Pools

Bob wants to sell 500 ETH for USDC over the next 5,000 blocks, or 0.1 ETH per block.

Charlie wants to sell 100 ETH for USDC over the next 2,000 blocks, or 0.05 ETH per block.

Until Charlie's order expires in 2,000 blocks, Bob's and Charlie's orders will be grouped together in a *pool*.

This ETH selling pool will sell ETH at a rate of 0.15 ETH per block for the next 2,000 blocks. Bob will get $0.1/0.15 \approx 66\%$ of the USDC the pool earns in this way. Charlie will get $0.05/0.15 \approx 33\%$ of the USDC the pool earns.

Virtual Orders

Every block for the next 2,000 blocks, the TWAMM has to buy 50,000 USDC worth of ETH on behalf of Alice and sell 0.15 ETH for USDC on behalf of the ETH selling pool.

We can imagine that the TWAMM splits each of these two sub-orders into trillions of tiny sub-sub-orders we call *virtual orders* (in fact, it breaks them into an infinite number of infinitesimal virtual orders).

The TWAMM then takes turns executing these virtual orders against its embedded AMM: first one of Alice's virtual orders, then one of the ETH selling pool's, then another of Alice's, and so on.

Arbitrage

Because Alice is buying much more ETH than the ETH pool is selling, the price of ETH on the embedded AMM will go up each block.

When this price is sufficiently high relative to the prices for ETH elsewhere, *arbitrageurs* will buy that cheaper ETH on other exchanges and sell it on the embedded AMM, bringing its prices back in line with the market average and ensuring good execution for Alice.

Order Expiry

After block 2,000, Alice's order will be completely filled, as will Charlie's. Bob's order to sell ETH is still in effect for the next 3,000 blocks, and the TWAMM will continue to execute it at a rate of 0.1 ETH per block during that period.

Barring any outside activity, this will push the price of ETH down on the embedded AMM over time, this time prompting arbitrageurs to bring prices back *up* once they get sufficiently out of line.

Economics

Because none of Alice, Bob, or Charlie are in a hurry to execute their orders, other market participants can infer that their orders represent less adverse selection than they otherwise would, and can provide them with low-price-impact execution.

Because the TWAMM will then be the best place for people like Alice, Bob, and Charlie to trade, LPs on the TWAMM's embedded AMM are likely to interact with a high volume of uninformed flow like theirs. This helps LPs earn money from fees while reducing their relative exposure to adverse selection.

Infinitesimal Virtual Orders

We mentioned above that the TWAMM splits long-term orders into infinitely many infinitely small sub-orders. There are two reasons to do this: smoothness and efficiency.

Smoothness

The primary goal of the TWAMM is to execute its long-term orders smoothly over time so that they are executed for close to the prevailing time-weighted average price.

As we decrease the size of the virtual trades, price movement on the AMM becomes less and less jagged.

In the limit, with infinitely many infinitely small trades, price movement is perfectly smooth as virtual trades are executed.

See https://github.com/para-dave/twamm/blob/master/splitting_exploration.ipynb (https://github.com/para-dave/twamm/blob/master/splitting_exploration.ipynb)

Efficiency

Because the TWAMM is designed to be used on Ethereum, explicitly calculating transactions for multiple virtual trades per block would be prohibitively expensive. However, when we have infinitely many infinitesimal trades, we can compute the outcome for traders in a single computation, no matter how many blocks it has been since we last checked.

Implementation

Lazy Evaluation

The TWAMM treats virtual sub-orders as if they take place in the space *between* blocks, which is important to avoid sandwich attacks.

To achieve this in a gas-efficient manner, the TWAMM uses *lazy evaluation*, computing the effect of virtual trades only when it is necessary to determine the outcome of an interaction.

Every time a user interacts with the TWAMM (for example, by trading with the embedded AMM or by adding a new long-term order), the TWAMM retroactively calculates the effect of all the virtual trades that took place since the last interaction.

Because these virtual trades are interacting only with the TWAMM's embedded AMM, the behavior of the TWAMM is completely deterministic between external interactions. Even if the TWAMM goes one million blocks between external interactions, the next time somebody interacts with it, it will be able to accurately calculate the results of all of the intervening virtual trades.

Front-ends plugged into the TWAMM will be able to account for virtual trades that have not yet been represented on-chain by keeping track of the current block number and doing the TWAMM calculation themselves.

Gas Optimizations

Pooling Orders

As demonstrated in the example, when we have multiple long-term orders in the same direction (i.e. selling ETH for USDC), we pool them together before splitting them into virtual orders. The TWAMM can then track balances using a version of the [billion-dollar algorithm (https://www.paradigm.xyz/2021/05/liquidity-mining-on-uniswap-v3/)](https://www.paradigm.xyz/2021/05/liquidity-mining-on-uniswap-v3/) used to track LP rewards in protocols like Compound and Uniswap.

Technically speaking, there are always two long-term order pools per TWAMM, one for each asset: for example, the pool selling USDC and the pool selling ETH. It just might be that one or both of these pools is empty at any given time.

Long-Term Order Expiration

One complication arises when using order pooling in conjunction with lazy evaluation.

Imagine Bob places an order to sell 100 ETH over the next 100 blocks, and Charlie places an order to sell 200 ETH over the next 200 blocks. Both orders are selling at a rate of 1 ETH per block.

Suppose nobody interacts with the TWAMM for the next 150 blocks, at which point a new external interaction occurs. For the first 100 blocks after Bob and Charlie placed their orders, their orders were pooled into a common order selling 2 ETH per block. However, for the 50 blocks after that, Charlie's order was on its own, selling only 1 ETH per block.

That means we have to do two separate trade calculations in order to find out what happened: one calculation for the results of the first 100 blocks, and one calculation for the final 50 blocks. In the worst case, if there had been orders expiring every block for the past 150 blocks, this would mean the TWAMM would have to process one trade per block, ruining gas efficiency.

The simplest workaround for this is to limit the number of blocks eligible for order expiry: for example, the TWAMM could specify that orders are only eligible to expire every 250 blocks, or roughly once per hour.

Canceling Long-Term Orders

Users can cancel long-term orders at any time. In practice, this allows users to select the cancelation time for their order down to the block. This does not increase the gas burden for the system since the user who wants to cancel pays their own gas.

Virtual Trade Math

Definitions

Assume it has been t blocks since the TWAMM last executed any virtual trades.

For the sake of simplicity, assume that no long-term orders have expired, so that the pool selling X was selling per block over the whole time period and the pool selling Y was selling per block over the whole time period.

Then the total amount of X sold over the period was and the total amount of Y sold over the period was .

Let's denote the embedded AMM reserves at the beginning of the time period as and , respectively.

Formulas

After all virtual trades are processed the embedded AMM will have X reserves of

where

From the constant product formula, we know

The pool selling X gets all of the Y that didn't end up in the embedded AMM — in other words,

and similarly,

Potential Attack Vectors

Sandwich Attacks

Description

In a sandwich attack (https://research.paradigm.xyz/MEV), an attacker, Atticus, sees that a trader, Trey, is about to do a trade on an AMM. Atticus sends two orders that *sandwich* Trey's, thereby extracting a profit from it.

Imagine Trey sends an order to buy ETH with USDC to an AMM. Seeing this, Atticus places an order to buy ETH on the AMM before Trey does, driving the price up. Because he is paying fees to the AMM and incurring price impact, Atticus is losing money on this order.

When Trey's order is filled, he buys the ETH at a higher price than he would have otherwise had to, because Atticus has pushed prices up. Trey's order drives the price up even higher.

Now, Atticus sells his ETH back to the AMM. Because Trey has pushed prices on the AMM up since Trey bought, he sells for more than he bought and is able to realize a profit.

This attack only makes sense for Atticus if he is able to guarantee that he will be able to sell his ETH back to the AMM immediately after Trey buys. Within a given block, this is possible if Atticus is a miner, makes a deal with one, or uses a service like Flashbots (https://medium.com/flashbots/frontrunning-the-mev-crisis-40629a613752).

Sandwiching and Virtual Orders

At first glance, virtual orders may seem especially vulnerable to sandwich attacks because everybody knows they are coming.

However, they are saved by the fact that they execute between **blocks. An attacker wishing to sandwich the TWAMM's virtual orders would have to trade against the embedded AMM at the end of one block, causing the virtual orders to execute at a bad price between blocks, and then trade in the other direction to close the trade at the beginning of the *next* block.

At present, there is no way for an attacker to guarantee that they will only trade at the end of a given block if they also get to trade at the beginning of the next block. When such *multi-block MEV* becomes more common, allowing traders to sandwich across multiple blocks, this may become more of an issue.

Information Leakage

The biggest tradeoff long-term traders are likely to encounter with the TWAMM is the *information leakage* they are exposed to when placing publicly visible orders, which are necessary due to the nature of Ethereum.

If a trader places a sufficiently large long-term order, other traders may be tempted to *front-run* it, buying up the asset on the TWAMM's embedded AMM and elsewhere in order to sell it back to the trader later as the long-term order pushes prices up.

Because users can cancel their long-term orders at any time, we expect overly aggressive front-runners to be exploited by other traders, keeping the overall impact of information leakage in check.

Example

Imagine Sally the spoofer has noticed aggressive front-running on the TWAMM. She buys one million USDC of ETH from a liquidity aggregator, pushing up prices across the market. Then she places a massive long-term order on the TWAMM to buy one hundred thousand USDC of ETH per block for the next 24 hours.

Immediately, Frank the front-runner sees this order and buys one million USDC of ETH through the aggregator, pushing up prices even further. Sally sells back her ETH through the aggregator for a profit, pushing prices back down and leaving Frank with a loss. Finally, she cancels her long-term order before any of it is filled.

Python Reference Implementation

You can see a Python reference implementation of the TWAMM here (https://github.com/para-dave/twamm).

This Jupyter notebook (https://github.com/para-dave/twamm/blob/master/twamm_demo.ipynb) demonstrates the behavior of the TWAMM with multiple offsetting long-term orders and arbitrageurs.

For the sake of simplicity, this Python version does not implement gas optimizations like pooled orders or true lazy evaluation.

Conclusion

We've sketched out the design of the TWAMM, but our work is just beginning.

If you are interested in working on this or similar problems, you can email dave@paradigm.xyz (mailto:dave@paradigm.xyz) or DM me on Twitter (https://twitter.com/messages/compose?recipient_id=1184231093576392704), or reach out to Uniswap Labs at ideas@uniswap.org (mailto:ideas@uniswap.org).

Acknowledgements: Sam Sun (https://twitter.com/samczsun), Georgios Konstantopoulos (https://twitter.com/gakonst), Michael Bently (https://twitter.com/euler_mab), Michael Kustermann, (https://twitter.com/MMKustermann)Kevin Pang (https://twitter.com/kevinxpang), Hasu (https://twitter.com/hasufl), Sam Bankman-Fried (https://twitter.com/SBF_Alameda), Henry Prior (https://twitter.com/priorupdates), Tom Cadwell (https://www.linkedin.com/in/tom-cadwell-404b49), Alex Wice (https://twitter.com/AWice), Mewny (https://twitter.com/mewn21), Big Magic (https://twitter.com/bigmagicdao), Lily Francus (https://twitter.com/nope_its_lily), Tarun Chitra (https://twitter.com/tarunchitra), Moody Salem (https://twitter.com/sendmoodz?lang=en), Noah Zinsmeister (https://twitter.com/NoahZinsmeister), Teo Leibowitz (https://twitter.com/teo_leibowitz?lang=en)

Written by: Dave White (https://www.paradigm.xyz/team/davewhite/), Dan Robinson (https://www.paradigm.xyz/team/danrobinson/), Hayden Adams (https://twitter.com/haydenzadams)

Important disclosures (https://www.paradigm.xyz/important-disclosures/)