SCALABLE INFRASTRUCTURE
FOR DIGITAL CURRENCIES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Geoffrey Ramseyer
September 2023

Preface

Recent years have seen a significant increase in interest among policymakers and financial institutions in digital currencies and a new generation of financial infrastructure. This thesis investigates the design of computational infrastructure and economic mechanisms for these new systems. Specifically, this thesis develops infrastructure that, unlike prior state-of-the-art systems, performs with computational scalability that is nearly-linear on both best-case and worst-case workloads, and that does not significantly limit the range of operations that a user can express. Building this infrastructure requires instantiating and therefore demonstrating the practical feasibility of previously-unused types of economic mechanisms for exchanging digital currencies, with efficiency and fairness properties of independent interest. This thesis then studies the economic tradeoffs encoded in mechanisms for exchanging assets, the tradeoffs involved when combining different mechanisms together, and ways in which these mechanisms interact with the underlying computational infrastructure.

Acknowledgments

This thesis would not exist without the kindness and support of so many people, to whom I owe an immense debt of gratitude. I have been extremely lucky to have many people generously take the time to teach me, for which I will always be thankful. First and foremost, I want to thank my advisors, Ashish Goel and David Mazières. Not only have I learned so much from the very different perspectives you bring to research problems, but you are two of the nicest people I have had the joy of meeting. The PhD is a long journey but I have always felt well-supported and it is impossible to leave a conversation with the two of you without coming away encouraged and enthusiastic, no matter what else is going on. And thanks to David for letting me defend this thesis even though I might not be quite ready to "survive on a desert island with nothing but an Arch Linux CD and maybe an Ethernet connection." Many thanks also to Dan Boneh and the rest of my defense committee, David Tse and Markus Pelger.

I also would like to thank Maryanthe Malliaris for teaching me all about ultrafilters and model theory while at the University of Chicago. This thesis does not directly touch on ultrafilters, but your approach to mathematics and to research has proved extremely helpful.

Many thanks to my fantastic coauthor Mohak Goyal, and all of the people from whom I had the chance to learn while at Stanford, whether by collaborating on a project, from so kindly taking the time to explain something interesting, or just through chatting at a group lunch: Brian Axelrod, Christie Di, Nikhil Garg, Lodewijk Gelauff, Jack Humphries, Zhihao Jiang, Anilesh Krishnaswamy, Evan Laufer, Paul Liu, Joachim Neu, Alex Ozdemir, Ben Plaut, Sukolsak Sakshuwong, Sahasrajit Sarmasarkar, Matthew Sotoudeh, Akshay Srivatsan, Nirvan Tyagi, Marco Vassena, Zachary Yedidia, Gina Yuan, Greg Zanotti, and Hongyang Zhang.

I also want to thank the people at the Stellar Development Foundation for a wonderful summer internship and especially Jon Jove, Graydon Hoare, Guiliano Losa, and Nicolas Barry for many fruitful conversations around the challenges of taking a research project into practice.

I first realized that I enjoyed working on open-ended research problems in middle school when some friends and I got together to start a FIRST Lego League team, The Battery-Powered Picklejar Heads [29]. I want to thank all of the Pickles for creating such a wonderful collaborative (if chaotic) environment, and I count myself lucky to consider you lifelong friends. I also would not be writing this without the invaluable guidance and support of the team-coaches-turned-lifelong-mentors, Paul Perry and Kevin Oye.

Thank you also to all of the friends from back home, from Chicago, and from California who have supported me and who make life fun. I would not have made it through the long PhD without the many wonderful times spent with all of you, whether backpacking, cycling, wine tasting (special thanks to the Stanford Wine Society), traveling somewhere together, or just enjoying a quiet afternoon. There are too many of you to list here, but know that I intend to thank all of you personally the next time I see you.

And last but certainly not least, thank you to my parents, Mark Ramseyer and Norma Wyse, and to my entire extended family, for supporting me and encouraging me for my entire life. Special thanks to my sister Jenny Ramseyer for being a fantastic partner-in-crime over many adventures and for always being there for me, and to my brother-in-law Eli Davis for being a wonderful addition to the family. The ideas behind some of the chapters in this thesis were cooked up while stuck in quarantine with you two in that tiny apartment.

Contents

List of Figures

Chapter 1

Introduction

Digital currencies have been the subject of much attention in recent years, for reasons both positive and negative. Digital infrastructure for financial transactions that is well-designed from the ground up could enable interoperability between the disparate computer systems in place today, reduce the costs of digital transactions across the economy, and increase access to financial pathways. At the same time, mania and investor excitement around so-called cryptocurrencies today has enabled widespread market manipulation, large-scale fraud, and outright theft. As such, regulators and policymakers today around the world are simultaneously pursuing experiments around new, digital financial infrastructure [253, 85, 256], and pursuing criminal charges against many cryptocurrency entrepreneurs.

These potential benefits are not merely the idle speculation of academics, but the subject of serious consideration and investment of resources by policymakers. In 2023 alone, for example, the United States Federal Reserve launched a new real-time payments service [253] and a digital currency pilot project [116], and the European Central Bank launched an upgrade to its earlier real-time payments system [85]. The European Union [84], the Bank of Japan [323], and many other nations and central banks [8, 252, 170, 254, 251] have similarly piloted or begun pilot central bank digital currency prototypes, and the People's Bank of China recently started a limited deployment of a central bank digital currency systems [256]. Private companies have similarly launched many private payment networks over the last several decades, and more recently have launched "stablecoins," [26, 293, 14] digital tokens on public blockchains pegged to the value of traditional currencies. In the words of the policymakers behind the central bank digital currency projects (such as in [256, 116, 252, 83]), the goals of these projects explicitly include improving the efficiency of financial transactions (including both domestic and cross-border payments), supporting competition between retail payment services, preserving open interoperability between payment systems (to e.g. maintain fungibility between deposits at different institutions), preserving the role of cash as a public good in the digital age, and expanding access to financial infrastructure.

Nor are any potential drawbacks mere abstract concerns. Also in 2023, the United States Government famously brought charges for several kinds of fraud against the founders of cryptocurrency exchanges and lending protocols, [53, 50, 48, 52], among others. The United States Securities and Exchange Commission is similarly pursuing charges of violations of securities laws against multiple cryptocurrency exchanges [49, 47]. [1] Fraudulent activity and the general bubble of speculation on this topic have led to serious financial harms.

These criminal activities are no small problem for market regulators, but even absent fraud and market manipulation, new digital financial infrastructure raises significant challenges for policymakers. For example, policymakers may be concerned that an influx of money into digital currencies may sap deposits from the traditional banking system. Citing precisely this concern, [2] a recent digital currency pilot project from the Bank of Japan [323], for example, tested the technical feasibility of various limits on digital currency holdings. Digital currencies have the potential to further exclude from modern economic life those without access to mobile devices and those less familiar with modern digital technology. Additionally, central banks may not want to engage in the business of providing banking relationships directly to consumers. Policymakers might also worry that opening access to financial infrastructure makes it easier for fraudulent and criminal activity to access financial systems, which could hamper their ability to target criminal resources. Simultaneously, citizens may worry that a digital currency that supplants physical cash could give governments unprecedented visibility and control over the financial lives of individuals, and even in a legal system with robust due process controls on such access, the possibility of cybersecurity breaches means that bad actors or adversary governments may access this valuable information.

All this is to mention that the decision to deploy new digital financial systems, or to allow and regulate private deployments of financial infrastructure, poses a host of policy questions fraught with complicated tradeoffs between effects spanning across society and the economy. As such, the form and functionality of any digital financial infrastructure is something that should be decided upon by policymakers and the democratic process, with consideration of the policy preferences of many different stakeholders.

In an ideal world, the one set of considerations that should not influence policy choices are the techological. Balancing between competing policy interests in the democratic process is already difficult enough. We should design infrastructure to satisfy a policy choice, not tailor our policy choices to the constraints of our infrastructure or to the specifics of legacy systems. Of course, some things are technologically impossible. We therefore seek to explore the Pareto-frontier of potential infrastructure for digital financial systems, so as to enable policymakers to make the technological choices that best match their policy objectives.

[1] There is no better summary of the case against Binance, one of the exchanges in question, than the words of its chief compliance officer, "[w]e are operating as a fking unlicensed securities exchange in the USA bro."

[2] " 経済的な設計の周辺機能は、銀行預金から CBDC への急激なシフトを招かないようにするための、各種セーフガードについて取り上げた" §2 [323], approximately translates as "The experiments tested the feasibility of many economic safeguards to guard against the rapid shift of money from bank deposits to CBDC holdings."

More importantly, we seek not only to illuminate policy choices but to design novel and improved system architectures and economic mechanisms for digital financial systems. We find that while not only does economic mechanism design influence computational architecture, but novel computational architectures can inspire novel economic mechanisms with independently interesting properties.

1.1 Thesis Objective

The focus of this thesis is on the technological infrastructure underpinning any potential new digital currency systems. Regardless of what policy choices are made, if such a system is to prove durable and of lasting, long-term societal value, the infrastructure should at least have the following properties.

1. **Scalability:** More computational resources should translate into higher throughput. We cannot know performance requirements a deployment will have, especially not one used many decades from now. Furthermore, policy choices and usage patterns necessarily influence infrastructure requirements. A system that infrequently moves large amounts of money between banks, for example, has very different latency and throughput requirements than one that frequently moves small amounts of money between retail users. Our infrastructure should be able to support whatever policies are chosen, and however many users and use-cases there may be.

   Each requirement, feature, and marginal user added to any system will slow it down, at least marginally. Computational scalability breaks this tradeoff. Instead of weighing features against performance, policymakers can weight features and performance against the required computational resources (such as operating costs). Low costs are better than high costs, but even high costs are better than impossibilities.

2. **Extensibility:** The system should seamlessly be upgradeable to accept new standards and components. We cannot know what all of the use-cases for a digital currency will be, and thus our infrastructure should be able to adapt and upgrade in response. New cryptographic standards may be adopted (e.g., post-quantum standards [58], or the recent explosion of zero-knowledge research [214, 212, 106, 213, 247, 173, 148, 187, 259, 107, 105, 91, 92, 61, 310, 98] [3]), new data formats and user identification systems may be needed, and new products and external integrations may need to be supported.

   Ideally, upgrades of different components should be deployable independently and in a self-service manner. Upgrading a small piece of a system is much easier than upgrading an entire system. The dual of this desideratum is **Robustness**; one badly-designed or faulty part of the system should not necessarily cause problems for the rest of the system.

[3]This is only a very incomplete list of very recent work on this topic.

Market regulators can (and should) maintain the ability to authorize new products and upgrades. However, in terms of actually getting a new idea into production in a large-scale piece of infrastructure that integrates with many different, independently operating banks and services, there is a huge difference between a system where one bank can, with authorization, unilaterally upgrade an implementation of one of its products, and a system where upgrades require implementation by market regulators themselves or by all other parties involved with the infrastructure. [4]

3. **Economic Efficiency:** This criterion depends on context, but broadly speaking, digital financial infrastructure should achieve the goals set out for it. Technical considerations of the infrastructure should also crucially not introduce misaligned incentives or opportunities for mechanical manipulation of economic outcomes. As an example of negative outcomes to avoid, existing public blockchains admit automatic manipulation of transaction ordering, which enables widespread front-running on decentralized exchanges [143, 161]. High latency and low throughput complicate cross-chain asset swaps, inadvertently requiring mechanisms to give out what amounts to free options [189], and require high transaction fees to limit transaction rates.

Towards our goals, the research of this dissertation is guided by the following set of principles.

1. **Every tradeoff should be the result of a fundamental technological impossibility.** This principle goes without saying in many contexts. We emphasize, however, that our goal here is to design systems from first principles on the Pareto-frontier of what is possible, and explicitly designate compatibility with legacy systems as a secondary concern. For example, when we design in chapter 2 Groundhog, a linearly-scalable execution engine for financial transactions, we design the semantics of the engine from a blank slate, rather than trying to modify the semantics of the engines used in many existing systems. As a result, Groundhog is not only linearly scalable but also flexible and upgradeable (as transactions execute arbitrary logic), and of course deterministic (and thus replicable and auditable).

   This principle also leads us to explore redesigning the semantics of higher-level applications, instead of taking existing application semantics as a hard constraint. In chapter 3, we design a high-performance currency exchange by changing how the exchange matches and satisfies trade requests. And finally, in chapter 4 we design protocols for sequencing transactions fairly when faced with the fundamental impossibility of an obvious global, uniform transaction order.

2. **The costs and benefits of tradeoffs should be transparent and well-understood.** Policymakers must be able to understand the effects of architectural tradeoffs in order to make

[4]Policymakers, bankers, and technologists all, quite naturally, have different areas of expertise. These should be used complementarily.

informed choices. For example, chapters 2 and 3 demonstrate that small amount of latency in transaction processing can be exchanged for a nearly-arbitrary amount of overall throughput. A common design choice in public blockchains is to amortize the latency of a consensus protocol by grouping transactions into blocks prior to invoking the consensus protocol. Our designs take advantage of this block structure by designing applications that can process whole blocks of transactions together (and computationally in parallel), instead of one-by-one. Chapter 3 also shows that in the case of a currency exchange, this manner of processing blocks of transactions together improves fairness of access and removes the problem of fragmenting liquidity between pairs of assets.

From the opposite angle, we study in chapter 5 the fine-grained economic tradeoffs when allocating capital for market-making in a widely-used, restricted class of automated strategies.

3. **Architectural decisions around tradeoffs should be as fine-grained as possible.** To reiterate, digital platforms for financial transactions are often systems that are shared between many stakeholders and subject to competing policy goals. As much as possible, each stakeholder should be able to make their own decisions about what architectural tradeoffs are most suited to their use-case.

   Concretely, our digital currency engine Groundhog simultaneously supports multiple consistency semantics, and is strictly more expressive and less restrictive than many systems deployed in the real-world today. There are even some things that are possible to do within Groundhog that are fundamentally impossible in traditional designs.

   Applications running within this engine can, for example, either achieve scalable throughput by (implicitly) processing whole blocks of transactions together, but applications that need to operate sequentially can do so, at the cost of limited throughput. Additionally, such applications can choose their own policy for sequencing transactions, and chapter 4 exhibits one potential policy.

1.2 Technical Overview

1.2.1 Replicated State Machines

Thus far, we have not precisely defined what digital financial infrastructure actually consists of, and have deliberately mixed together terms like "digital asset," "central bank digital currency," and "cryptocurrency." These terms mean entirely different things in real-world applications, and are not, in general, interchangeable. One USD deposited in a bank is legally distinct [5] from one USD's worth of liabilities issued by the US government and from one USD's worth of digital or physical cash,

[5] I am not a lawyer, and this is not legal advice.

and is even distinct one USD deposited in a different bank. All of these carry completely different significance from cryptocurrencies like Bitcoin [249] and Ethereum [308], which are again completely different from ever more obscure (and very often dubious) financial products [37, 31, 51, 139, 305]. [6]

We mixed these terms introducing this dissertation because the differences between these objects are, to a first approximation, unrelated to the subject of our research. All of these objects are represented as bits within a digital system, and ownership changes or other financial transactions reduce to changing bits according to predefined sets of rules. The very real differences between these asset classes comes from the meanings that are assigned to the bits, not the bits themselves. [7] Different assets and different deployment contexts create different transaction patterns and infrastructure requirements, but nevertheless, nearly all digital financial infrastructure operates under some variant of the same underlying architectural paradigm.

That common paradigm is the *replicated state machine*. The infrastructure maintains some *state*; for example, the state of a digital currency system might contain an account balance for each user and configuration data. Users submit *transactions*, which modify state in some specified manner; for example, a transaction might verify a digital signature, and then transfer money from one account to another. For robustness against hardware failure, this state machine is often *replicated* across multiple physical machines; one common configuration is one primary replica with several backup replicas [230, 299, 257, 57], each of which is ready to take the place of the primary.

For this paradigm to be self-consistent, each copy of the state machine maintained by a backup replica has to be identical to that maintained by the primary. A core challenge for this paradigm is in maintaining this consistency between replicas as the state machine evolves over time, in response to user transactions. The most straightforward approach, used in a wide variety of applications including not just digital currency prototypes [249, 64, 234] but also networked filesystems, distributed databases, and many others [257, 299, 230, 114], is to break the challenge into two pieces. First, replicas use some *consensus protocol*[8] to agree on a totally ordered log of transactions, where log entries are only appended to the end of the log and the log grows monotonically over time. And second, each replica sequentially executes the transactions in the log, according to the agreed-upon order, to update the state machine over time.

As long as each transaction can be executed deterministically [9] and transactions execute in the

[6] Which are all again different assets like so-called "non-fungible tokens" [159] that purport to represent ownership of art objects or to prove identity [306]. One wishes Walter Benjamin's treatise on art and mechanical reproducibility were more widely read [93].

[7] Digital financial infrastructure is entirely digital, without direct ability to manipulate any physical objects, except in as much as the physical world grants it the ability to do so by means of agreement that changes in the digital system should be reflected in the physical world. To borrow Saussure's philosophy of language [279], social consensus creates a relationship between the "signifier" bits and ownership of the "signified" assets.

[8] Such as one of [222, 316, 242, 181, 274, 198, 257, 114]. Different protocols have different performance characteristics and operate in different contexts.

[9] To be pedantic, what is strictly necessary is that a transaction's execution is the same in each copy of the state machine. Randomization would be possible if every replica has access to the same (pseudo)random bits. For example, replicas could start with a shared seed for a pseudorandom generator, or a primary replica could insert the random

same order at each replica, then every replica of the state machine will compute the same state (after any finite index in the log). This decomposition has the convenient property that the computational work of executing transactions need not proceed strictly in lock-step between all replicas, as it suffices for replicas to simply agree on a transaction log. Similarly, this computational work is not on the critical path between receiving a transaction request (e.g., from a client) and confirming that the transaction is added to the log.

At first glance, this sequential model is simple to reason about and a straightforward foundation upon which to build other applications (like digital currencies). However, this simplicity belies some fundamental flaws and abstracts away details of the infrastructure that can have a profound impact on the overall usefulness of an application. In particular, the architecture suffers from poor computational scalability, and financial applications suffer degraded economic performance due to a mismatch between this model and how information actually flows through the system. This thesis, therefore, investigates the interconnections between computational performance, economic performance, and the ordering semantics (and information flow) of replicated state machines. Crucially, it is the study of these aspects *together*, and not in isolation, that gives rise to systems, such as SPEEDEX in chapter 3, which can simultaneously improve both the computational scalability and the economic performance over systems built on the sequential replicated state machine model.

1.2.2 Scalability and Computational Performance

Unfortunately, the sequential architecture introduces a fundamental performance limitation. While the speed of computer hardware has historically grown faster every year, a CPU can only execute a finite number of operations in a finite amount of time. Increasing the throughput of any computer system beyond the limit of any one CPU requires using many CPUs together. As Moore's law nears its (predicted) end, the ability of any computer system's design to scale its performance with the number of CPUs available becomes increasingly important. In particular, financial infrastructure may need to last for a long time and over time may need to be upgraded with additional features and support more and more users transacting more and more frequently, which makes scalability all the more important. While much work has been devoted to accelerating the performance of sequential transactions using techniques like speculative execution [120] optimistic concurrency control [179, 309], and even reordering transactions to better suit these techniques [294], if any change to the order of transactions affects the final state of the system, then it remains fundamentally impossible to perform better than just sequentially executing transactions on one CPU.

And even when these techniques perform well, the computational overhead of these techniques becomes a limiting factor even under best-case scenarios, meaning that systems cannot take full advantage of the available CPU hardware [234, 179]—see e.g. Fig. 2.5 plotting the performance of [179] on a payments workload, as compared to our linearly-scalable system in chapter 2 on that

bits it uses to execute a transaction into the log.

workload in Fig. 2.4. For critical infrastructure like a digital currency platform, slow performance on some transaction patterns is not merely an inconvenience but a potential security threat. Worst-case patterns might conceivably be exploited by an adversary to slow down the system, and could occur naturally when, for example, a large company pays its employees or around tax payment deadlines. [10]

Unfortunately, this limitation, that blockchain architectures based on replicated state machines have to be slow and offer only limited throughput has unfortunately become accepted as "conventional wisdom," and acts as a justification for high transaction fees, complicated architectural tradeoffs, and other negative externalities. We demonstrate in this thesis that this so-called wisdom is false. Specifically, chapters 2 and 3 build near-linearly scalable systems to support digital currencies in the replicated state machine paradigm.

Our systems achieve this scalability by making two changes to the sequential paradigm. First, we change how transactions modify the state machine. Many existing systems attempt to increase performance in this manner, such as by dividing a user's money into multiple distinct units [249, 234, 84, 116] or dividing the state machine into several distinct sub-machines [38, 40, 10, 11]. Unfortunately, on their own, these designs provide only limited scalability (while complicating system architecture) [11] or severely limit expressivity. [12] Systems focused only on payments sometimes move activity into privately-run "payment channels" [263], but incur significant (and nontrivial to quantify) economic costs [268, 183]. [13]

And secondly, we change what it means for transactions to be ordered in the state machine's log. Instead of computing a total ordering on all transactions, our systems in chapters 2 and 3 compute only a partial ordering. When two transactions are not ordered relative to each other, our systems execute these transactions in parallel without any sequencing coordination between CPU cores. Given the nondeterminism of thread scheduling and relative CPU performance, this means the transaction pair is ultimately executed in a *nondeterministic* ordering (and may interleave the two transactions). Done naively, this nondeterminism would obviously have the potential to create nondeterminism in a state machine, leading to divergence between state machine replicas.

[10]For example, suppose that a digital currency system records amounts in accounts, and every transaction includes a read-modify-write operation, such as a withdrawal, on the same account. Then this transaction distribution is a worst-case scenario for optimistic concurrency control.

[11]In such a design, the fact that each component state machine can execute transactions separately provides opportunities for computational parallelism, although each component still has the same performance limitations. Additionaly, these designs substantially complicate cross-component transactions, which adds (often substantial) latency. In the case of a digital currency system, these designs make money not entirely fungible, as money in one component is no longer the same as money in another.

[12]The commonly used "Unspent Transaction Output" (UTXO) model of [249, 234, 84, 116] divides money into discrete "coins", each of which is constrained to only be spent once. This constraint makes it extremely difficult to coordinate modifications to shared state between users without an out-of-band communication channel. Done naively, any users might modify one object, but only one will succeed, and the rest must wait until the block containing the successful transaction is published before retrying. More complex systems require dedicated external coordination systems with complicated incentive structures, as in, e.g., [218].

[13]As well as significant architectural complexity, security risks [287, 96], and a more complicated user experience [80, 266].

However, what we show is that combining these two ideas together leads to system designs that are both linearly scalable (through computational parallelism) and deterministic (and therefore replicable), without many of the drawbacks of excessively constrained state models. Specifically, the key insight is to design a set of transaction semantics so that transactions which are not ordered relative to each other are semantically *commutative*. That is, no matter the nondeterminism of a replica's thread scheduling and other internal operational details, the end result of applying a partially ordered log of transactions is still deterministic.

To demonstrate this potential of this architectural paradigm, chapter 2 develops a set of general-purpose commutative transaction semantics that not only is strictly more flexible than the semantics used in several nominally general-purpose cryptocurrency systems [40, 38], but also admits functionality that fundamentally cannot be implemented in the traditional sequential model. [14] Many but not every application can be implemented effectively in this commutative model, but these semantics admit a fallback to sequential execution. As such, each application built within Groundhog can choose on a fine-grained level what level of ordering between operations that it requires, with a corresponding tradeoff in performance of that application.

In order for digital currencies to effectively support cross-border payments, users need to be able to easily trade one currency for another. But traditional designs of exchange systems for currencies (or trading systems for other assets) have inherently noncommutative semantics, and would not be implementable on top of Groundhog without the (throughput-limited) fallback to sequential execution. In chapter 3, therefore, we redesign the semantics of a exchange to build SPEEDEX, an exchange for digital currencies that operates at near linear scalability in the same commutative model as in Groundhog.

1.2.3 Ordering and Information Flow

The partial ordering that both Groundhog and SPEEDEX use is the ordering that arises directly out of how the overall replicated state machine paradigm operates. Each system uses some kind of consensus protocol to ensure every replica's transaction log is (eventually) identical, but these consensus protocols can be resource-intensive and may often require multiple network round-trips between replicas, adding considerable latency to transactions. As such, most systems amortize the cost of consensus by grouping transactions into blocks, and then invoking consensus on these blocks (and not on each transaction individually). This means that most systems append transactions to the log in blocks at discrete times (hence the term "blockchain"). As such, the partially-ordered design paradigm of Groundhog and SPEEDEX is not removing ordering information from the system, but

[14]Groundhog executes many transactions in parallel, instead of sequentially. As such, if one transaction stalls, the systems can pause execution of that thread and move to other productive work. By contrast, if transactions were executed sequentially, results of later transactions could depend on the stalled transaction.

This ability to stall allows transactions to call into external services and systems that might not return results quickly. We use this feature to solve the so-called "free option" problem in §2.7.

rather, merely refusing to artificially fabricate information not already present in the architecture.

This observation points towards a more fundamental question: what do ordering and time mean in a distributed system? In a system architecture that commits transactions in blocks, information about what transactions are committed, and therefore what the current state of the system is, can only propagate to downstream clients at discrete intervals. In other words, the most recent state of the state machine that a user might observe, i.e., prior to sending a transaction, is always a state reached after applying an entire block of transactions, and not an intermediate state after only some transactions in a block are executed.

Groundhog's semantic design, therefore, reflects the information that users actually would possess in a real-world instantiation, and indeed this alignment between semantics and real-world information flow prevents a class of security issues present in systems that execute transactions sequentially. Furthermore, by eliminating the need to fabricate an ordering between transactions within a block, this design paradigm eliminates the ability of any actor in the system to deliberately manipulate that fabricated ordering for their own benefit. This type of manipulation, often called "front-running," is widespread in existing decentralized exchanges [143, 265] and can be highly profitable, at the expense of other users receiving worse exchange rates. [15] In contrast, SPEEDEX, by the mere fact of not ordering the transactions within a block, entirely eliminates this class of front-running. To reiterate, these application benefits, that result from aligning application semantics with the overall system design, are entirely independent of the performance benefits that this architectural paradigm grants.

Alternatively, some applications might need process transactions in a strictly total ordering. There thus arises a need to be able to recover a total ordering in a well-founded manner from a distributed system that does not directly provide one. The core problem is that information spreads through a distributed network non-uniformly. That is to say, every position in the network—every replica, and every network observer—will receive new transactions in a (possibly) different order. The front-running mentioned above is particularly rampant on today's decentralized exchanges precisely because at any given time, one agent has sole, dictatorial authority to determine the ordering of new transactions.

However, any system designer that wants to avoid giving any one agent dictatorial control over the transaction ordering, while maintaining a set of basic properties of the output ordering [16] quickly runs into Arrow's Impossibility Theorem [76]. As we show in chapter 4, this problem is a "streaming" instance of the classic problem of social choice theory. The distributed system must order a countably infinite set of transactions Instead of a finite set of "candidates," and must do so with only partial information at any point in time. This perspective on the problem enables us to

[15]An analogous phenomenon occurs in traditional markets, where market participants compete for low-latency access to exchanges [103].

[16]A basic set of properties might naturally include *unanimity* (if every part of the distributed system sees transaction A before transaction B, then A should be sequenced before B) and *independence of irrelevant alternatives* (how transaction C propagates through the system should not affect whether A is sequenced before B).

study the problem in a well-founded theoretical manner and adapt classical ranking algorithms into our streaming setting.

1.2.4 Economic Performance

Not only do computational concerns constrain the implementation of economic mechanisms, [17] but in fact, the novel partially-ordered paradigm discussed above actually enables the design and efficient implementation of new economic mechanisms with new, useful economic properties. Chapter 3 develops an exchange, SPEEDEX, for digital currencies in this paradigm. Because SPEEDEX executes large batches of transactions together, and not individually, the exchange need not myopically focus on an individual trade and can compute at a higher level how to most efficiently clear the entire market. To be precise, instead of myopically computing a (varying) exchange rate for each trade request, SPEEDEX is able to offer every trade request in a batch the same exchange rate. An exchange rate where the buy volume equals the sell volume is computable efficiently precisely because SPEEDEX can access all of the information of a batch simultaneously. In fact, such a mechanism is required to provide transaction commutativity, and reflects the fact that every trade request in one batch is committed to the underlying replicated state machine at the same time.

Moreover, this paradigm enables SPEEDEX to work not just with isolated pairs of currencies but with many currencies simultaneously. Instead of fragmenting market liquidity between specific trading pairs, SPEEDEX offers an exchange rate between every pair, and ensures that these exchange rates are free of cyclic arbitrage. [18] Rather than explicitly routing trades through intermediate assets (as one might have to in a traditional design, if a given trading pair is not listed or illiquid), SPEEDEX, in effect, automatically routes trades through all available liquidity.

Additionally, this mechanism eliminates the need for ultra-low-latency access to the exchange; all users see the same exchange rates, so long as a user's network latency is sufficiently shorter than SPEEDEX's batch frequency ($\sim$ seconds—milliseconds matter, but not microseconds). This fairness property would be important for a system used for cross-border payments by users in widely-separated geographic locations, and it has been argued that under some circumstances, such systems reduce the cost of liquidity provision [103]. And as mentioned above, because every trade in a batch sees the same exchange rates, the type of risk-free front-running, powered by advesarial ordering manipulation that is widespread on existing decentralized exchanges, is impossible.

Implementing SPEEDEX requires computing in every batch the solution to an optimization problem (the equilibrium of a linear instance of an Arrow-Debreu exchange market [78]). In theory, solutions can be approximated in polynomial time, but developing algorithms that can run efficiently

[17]Perhaps the canonical example of this interaction between economic design and computational tractability is the combinatorial auction, famously used for allocating wireless spectrum [275].

[18]Within a batch, no user could profit or lose money by sending a set of trades along a cycle of assets. For any sequence of assets $\mathcal{A}_1, ...\mathcal{A}_n$, the product of the exchange rates $r_{\mathcal{A},\mathcal{B}}$ along this cycle $r_{\mathcal{A}_1,\mathcal{A}_2} \times \cdots r_{\mathcal{A}_{n-1},\mathcal{A}_n} \times r_{\mathcal{A}_n,\mathcal{A}_1} = 1$.

in practice, even as the number of market participants grows, requires taking advantage of additional market structure present in our problem instances. Namely, that while the number of people in the world is large, the number of currencies is relatively small. Additionally, cross-border payments trade one currency into one other, so the linear utility functions in the exchange market have positive marginal utility for only two assets. SPEEDEX also uses a linear program to shape the approximation error of the theoretical algorithms into forms (such as transaction fees) acceptable for real-world contexts, and this same piece of additional market structure makes this linear program small and empirically tractable.

And from the opposite viewpoint on this topic, chapter 5 studies the precise economic tradeoffs embedded in a class of exchange mechanisms that are explicitly designed to require as little computation as possible to operate. Minimizing compute requirements is especially important in a system that executes transactions sequentially (as such systems lack scalability, as discussed above). [19] These mechanisms are parametrized by a choice of "trading curve," each choice of which implicitly encodes tradeoffs in economic performance. Chapter 5 studies these tradeoffs and gives an explicit convex program that characterizes, for any scenario, the optimal choice of curve. This program additionally is analytically tractable, which means that it can explain precisely the informal rationale behind the curves chosen in practice.

Finally, chapter 6 studies the interaction between batch exchanges like SPEEDEX and CFMMs. These two market design innovations operate in different economic contexts and computational paradigms. For example, CFMMs are market-makers, and traders interact directly with CFMMs, but in a batch exchange, traders interact with each other, and with no market-maker. CFMMs trade immediately using a predictable pricing mechanism, while batch exchanges settle after a short time (with less price predictability). In a batch exchange, a CFMM's curve can act akin to an efficient parametrization of a (short-term) market-making strategy. As such, there is not an obviously correct method for integrating the two mechanisms, and in fact different batch exchanges have made qualitatively different choices. These different choices lead to different economic outcomes.

1.3 Thesis Outline

Chapter 2 presents the design of Groundhog, a replicated state machine designed to provide a scalable infrastructure platform for a central bank digital currency. Transactions execute arbitrary smart contracts, which provides the system with flexibility and enables seamless upgrades over time.

The novelty of Groundhog's design is that it concurrently executes blocks of commutative transactions. Unlike prior work, transactions within a block in Groundhog are not ordered relative to one another. Instead, our key design insights are first, to design a set of commutative semantics

[19] Furthermore, market-making mechanisms, such as the ones studied here, tend to require read-modify-update operations on a single data structure, which is the worst case scenario for techniques for accelerating sequential workloads mentioned in §1.2.2.

that lets the Groundhog runtime deterministically resolve concurrent accesses to shared data. Second, some storage accesses (such as withdrawing money from an account) conflict irresolvably; Groundhog therefore enforces validity constraints on persistent storage accesses via 2-phase locking when assembling new blocks of transactions.

These two ideas give Groundhog a set of semantics that, while not as powerful as traditional sequential semantics, are flexible enough to implement a wide variety of important applications, and are strictly more powerful than the semantics used in some production blockchains today.

Unlike prior smart contracting engines, transactions throughput never suffers from contention between transactions. Using 96 CPU cores, Groundhog can process more than half a million payment transactions per second, whether between 10M accounts or just 2.

This chapter is based on joint work with David Mazières [271].

Chapter 3 presents the design of SPEEDEX, a decentralized asset exchange designed to operate in the partially-ordered, replicated state machine paradigm described in Chapter 2.

SPEEDEX offers several advantages over prior decentralized exchanges. It achieves high throughput—over 200,000 transactions per second on 48-core servers, even with tens of millions of open offers. SPEEDEX runs entirely within a Layer-1 blockchain, and thus achieves its scalability without fragmenting market liquidity between multiple blockchains or rollups. It eliminates internal arbitrage opportunities, so that a direct trade from asset $\mathcal{A}$ to asset $\mathcal{B}$ always receives as good a price as trading through some third asset such as USD. Finally, it prevents certain front-running attacks that would otherwise increase the effective bid-ask spread for small traders.

SPEEDEX's key design insight is its use of an Arrow-Debreu exchange market structure that fixes the valuation of assets for all trades in a given block of transactions. We construct an algorithm, which is both asymptotically efficient and empirically practical, that computes these valuations while exactly preserving an exchange's financial correctness constraints. Not only does this market structure provide fairness across trades, but it also makes trade operations commutative and hence efficiently parallelizable. SPEEDEX is prototyped but not yet merged within the Stellar blockchain [233], one of the largest Layer-1 blockchains.

This chapter is based on joint work with Ashish Goel and David Mazières, published at the 20th Usenix Symposium on Networked Systems Design and Implementation in 2023 [269].

Chapter 4 studies the question of "fairly" ordering transactions in a replicated state machine. Each replica receives transactions over a network from clients in a possibly different order, and the system aggregates these orderings into a single order. This problem is akin to the classic problem of social choice, where rankings on candidates are aggregated into an election result.

Two features make this problem novel and distinct. First, the number of transactions is unbounded, and an ordering must be defined over a countably infinite set. And second, decisions must be made quickly and with only partial information. Additionally, some faulty replicas might alter their reported observations; their influence on the output should be bounded and well understood.

We specifically focus on the Ranked Pairs method. Analysis of how missing information moves through the algorithm allows our streaming version to know when it can output a transaction. Manipulation of a tiebreaking rule ensures our algorithm outputs a transaction after a bounded time (in a synchronous network).

Prior work proposes a "γ-batch-order-fairness" property, which divides an ordering into contiguous batches. If a γ fraction of replicas receive a transaction tx before another transaction tx', then tx' cannot be in an earlier batch than tx.

We strengthen this definition to require that batches have minimal size, which must be handled carefully in the presence of faulty replicas. This gives the first notion of order-fairness that cannot be vacuously satisfied by arbitrarily large batches and that is satisfiable in the presence of faulty replicas. Prior work relies on a fixed choice of γ and bound on the number of faulty replicas f, but we show that Ranked Pairs satisfies our definition for every γ simultaneously and for any f, where fairness guarantees linearly degrade as f increases.

This chapter is based on joint work with Ashish Goel [267].

Chapter 5 studies a class of exchange markets, known as Constant Function Market Makers (CFMMs). These mechanisms have been deployed widely in prediction markets, but have recently become especially prominent in the modern Decentralized Finance ecosystem. The reason is that their operation requires very little computation; this feature is highly atractive in the highly computationally-constrained (sequential) environments of many public blockchains. A CFMM is parametrized by a trading "curve,"[20] which defines which trades the CFMM will accept. Each curve encodes a different set of accepted trades and thus a different set of economic tradeoffs; how should a CFMM operator choose one curve over another?

We show that for any set of beliefs about future asset prices, an optimal CFMM trading function exists that maximizes the expected fraction of trades that a CFMM can settle. We formulate a convex program to compute this optimal trading function. This program, therefore, gives a tractable framework for market-makers to compile their belief function on the future prices of the underlying assets into the trading function of a maximally capital-efficient CFMM.

Our convex optimization framework further extends to capture the tradeoffs between fee revenue, arbitrage loss, and opportunity costs of liquidity providers. Analyzing the program shows how the consideration of profit and loss leads to a qualitatively different optimal trading function. Our model additionally explains the diversity of CFMM designs that appear in practice. We show

[20]The level set of an eponymous trading function.

that careful analysis of our convex program enables inference of a market-maker's beliefs about future asset prices, and show that these beliefs mirror the folklore intuition for several widely used CFMMs. Developing the program requires a new notion of the liquidity of a CFMM, and the core technical challenge is in the analysis of the KKT conditions of an optimization over an infinite-dimensional Banach space.

This chapter is based on joint work with Mohak Goyal, Ashish Goel, and David Mazières, published at the 24th ACM Conference on Economics and Computation in 2023 [185].

Chapter 6 studies the interaction between CFMMs and batch exchanges. Different real-world implementations have used fundamentally different approaches towards integrating CFMMs in batch exchanges, and yet different approaches lead to different economic and computational tradeoffs.

We first provide a minimal set of axioms that express the well-accepted rules of batch exchanges and CFMMs. These are asset conservation, uniform valuations, a best response for limit orders, and non-decreasing CFMM trading function. In general, many market solutions may satisfy all of these axioms.

We then describe several economically useful properties of market solutions. These include Pareto optimality for limit orders, price coherence of CFMMs (as a defence against cyclic arbitrage), joint price discovery for CFMMs (as a defence against parallel running), path independence for simple instances, and a locally computable response of the CFMMs in equilibrium (to provide them predictability on trade size given a market price). We show fundamental conflicts between some pairs of these properties. We then provide two ways of integrating CFMMs in batch exchanges, which attain different subsets of these properties.

We further provide a convex program for computing Arrow-Debreu exchange market equilibria when all agents have weak gross substitute (WGS) demand functions on two assets – this program extends the literature on Arrow-Debreu exchange markets and may be of independent interest.

This chapter is based on joint work with Mohak Goyal, Ashish Goel, and David Mazières [270].

Chapter 2

Groundhog: Scaling Smart Contracting via Commutative Transaction Semantics

2.1 Introduction

Blockchains offer many attractive features such as resiliency, transparency, and, most importantly, the ability to transact atomically across mutually distrustful parties. A blockchain could potentially act as a global cross-service coordinator, allowing atomic transactions across arbitrary databases that were not explicitly built to work with each other. Smart-contracts provide extensibility as revolutionary as JavaScript in the browser or apps on phones, and have had a huge multiplier effect on innovation by attracting more developers with self-serve deployment. But are we ever going to see blockchains used for significant payment volume and throughput? What would it take for a central bank or consortium of commercial banks to adopt a blockchain for fiat currency payments?

One realistic path for mainstream adoption is to implement a wholesale central bank digital currency (wCBDC). A wCBDC would connect banks and other financial institutions to each other and to power-users and developers, which would lower the bar for innovation without the need for average citizens to manage crypto wallets or the threat of liquidity leaving our fractional reserve banking system. Unfortunately, existing blockchains offer inadequate payment-per-second rates for such a critical application. Worse than actual throughput numbers, blockchains have poor asymptotic scaling—they derive limited benefit from more CPU cores, particularly in the worst case of high contention ([179, 174, 278]). Designs that divide state between "shards" [9, 321, 307, 291] or "roll-ups" [263, 203, 18, 10, 262, 11, 35, 183] accelerate only particular payment patterns and complicate any cross-shard transaction.

A core challenge is that most blockchains are structured as deterministic replicated state machines, which in existing systems has hindered expressiveness, the use of hardware parallelism, or both. Outside the blockchain context, transaction processing typically exploits hardware parallelism by using locks for serialization in case of transaction contention.

Unfortunately, if transaction results depend on ordering (e.g., the second of two \$10 payments from an account with only \$15 must fail), then inter-thread scheduling will introduce non-determinism in transaction results, replicas will diverge, and the blockchain will fail.

Specifically, a state machine implementing a wCBDC must display at least the following properties.

- **Deterministic Operation** Replicas of the state machine must stay in consensus, and furthermore, past transactions must be replayable to enable auditing of historical data.
- **Flexibility** The state machine should support executing arbitrary, untrusted computations, so as to support as-yet-uninvented applications, data formats, and protocols.
- **Scalability** Any significant piece of financial infrastructure is extremely difficult to modify once deployed. A wCBDC design should be able to scale to transaction volumes far beyond those seen today.
- **Robustness** No malicious smart contract, nor any bug in any contract, should be able to degrade system performance or reduce transaction throughput.

Our Contribution: Groundhog This paper presents Groundhog, a deterministic smart-contract wCBDC ledger designed to achieve high and scalable worst-case performance. Groundhog introduces a new point in the design space: totally ordered blocks of unordered transactions. To execute a block of unordered transactions deterministically, we must ensure they commute—i.e., have semantics unaffected by execution order (which enables efficient parallelization [125]). Groundhog achieves commutativity by executing every transaction in a particular block on the identical snapshot of the ledger state at the start of the block. It collects and defers transaction side-effects, most of which are commutative (e.g., increment an integer) and can be combined independently in the local memory of each CPU before being applied to the global ledger state at the end of execution.

Of course, smart contracts do sometimes need sequencing—e.g., to avoid double-spending a token balance. Groundhog shifts the burden of synchronization from the ledger to smart contracts, which must ensure that transactions that require sequential execution cannot be included in the same block. More generally, contracts can dynamically provide to Groundhog a set of constraints on ledger state, and replicas, when proposing a block of transactions, ensure that these constraints are always satisfied. A process akin to 2-phase locking makes the block assembly problem efficient and scalable.

A few simple abstractions in the API make this much simpler than it sounds. For instance, Groundhog provides non-negative integers to which contracts can commutatively add and subtract values. Conceptually, one can think of the subtractions happening at the beginning of the block and the additions at the end. A set of transactions *conflicts* and cannot be included in the same block if the subtractions would ever bring the integer below zero. These non-negative integers are a perfect primitive to represent account balances and semaphores. Groundhog also provides simple byte strings (where two writes of different values conflict) and ordered sets (where duplicate insertions conflict).

Although these snapshot semantics are not as general as strictly serializable semantics (as Groundhog cannot execute arbitrary ordered transactions in the same block), we are nonetheless able to implement all of the popular smart contract types we tried, including payments, auctions, an automated market maker, and a partially-collateralized algorithmic money-market protocol. Implementing these applications in Groundhog typically requires only small API changes. As such, we expect most users to not use Groundhog's APIs directly, but rather, to write contracts leveraging higher-level application-specific APIs.

Groundhog's semantics are strictly more powerful than the asynchronous message-passing architectures implemented in some large-scale blockchains [40, 38]. Furthermore, Groundhog does not require sharding or otherwise subdividing the blockchain's state to achieve its parallel execution, does not require precomputation of transaction writesets, and does not rely on any manual analysis of untrusted smart contracts.

As an additional benefit compared to other blockchains, Groundhog eliminates the ability for block producers to extract value by ordering the transactions within a block, a widespread phenomenon in current systems [143, 265] (although a malicious replica can still delay transactions to a future block). Our evaluation of Groundhog shows that it scales nearly linearly to 96 cores (the most we have available) no matter the level of contention between transactions, and handles more than half a million payments per second between 10M accounts.

2.2 Architecture Overview

Groundhog is a deterministic, scalable execution engine for a replicated state machine, and consists of the following components.

- *Smart contracts*, implemented as WebAssembly modules, are deployed at 32-byte *addresses*.
- Each contract may possess some local storage, which consists of *typed objects* (§2.3.2). Each object is referenced by a 32-byte identifier in the contract's namespace (so each object has a unique 64-byte identifier).

- Each *transaction* is a function call on some smart contract, specified by a contract address, a method number, a bytestring input for the function, and some metadata. Contracts may call into other contracts arbitrarily.

What makes Groundhog unique is the way that it processes blocks of transactions. Instead of executing transactions sequentially, Groundhog makes a snapshot of the entire state machine before processing any transactions in a block, and then executes each transaction in the block against this snapshot (§2.3.1). Transactions in Groundhog are only partially ordered, and there is no ordering between transactions in the same block. Note, however, that the contract and storage addressing scheme and cross-contract call semantics are the standard design for many real-world blockchains (e.g. [308, 41]).

Write Conflicts A core challenge in the design of Groundhog is in managing the write conflicts that result from this novel consistency guarantee. The Groundhog runtime carefully tracks the side-effects of each transaction as a set of typed, commutative modifications to persistent storage. Because contracts provide the Groundhog runtime with this small amount of semantic information, Groundhog can deterministically resolve multiple modifications in one block to the same object. For example, a contract transferring money would instruct the runtime to add to and subtract from various accounts, and addition operations form an Abelian monoid.

Object Constraints However, some financial operations may semantically conflict, and as such many smart contracts need to maintain constraints on their state. For example, a digital cash system might require that every user's account balance be nonnegative. Groundhog allows contracts to explicitly express a set of constraints on their local storage. For example, the Groundhog runtime offers a “nonnegative integer” object type in a contract's local storage (§2.3.2), to which a transaction can add and subtract; the runtime, by conservatively excluding transactions when assembling a block, ensures that the value of the integer is nonnegative after applying the block of transactions. In a traditional digital currency system with sequential semantics, if two transactions spend the same coin, one transaction would succeed while the second would fail; instead, Groundhog ensures that a block contains no such conflicting pairs.

Deployment Context We envision Groundhog deployed within a large-scale digital currency system, where each replica is operated by a central bank or a large financial institution—institutions with the resources to deploy high-performance systems with many CPUs. These replicas would be connected with some consensus protocol. Groundhog is agnostic to the choice of protocol, but is designed around a context where a leader periodically (once per second, perhaps) proposes a block of transactions, such as in SCP [233], HotStuff [316], or BA$\star$ [181]. Proposed blocks would then be checked for correctness (that is, the absence of conflicting transactions) by other replicas.

Repeated proposal of invalid blocks is a potential denial of service vector for a system using Groundhog. In this context, we imagine that replica operators themselves can be sanctioned if they repeatedly violate the protocol by proposing invalid blocks. However, end users and contract authors may be anonymous or difficult to trace, and are potentially malicious. Our implementation of Groundhog allows an honest replica to include potentially malicious transactions executing untrusted smart contracts in a block without compromising block validity.

2.2.1 Design Properties

Our design for Groundhog meets our requirements for the execution engine of a digital currency system.

Deterministic Operation The output of each transaction is a set of modifications to persistent storage. Groundhog ensures that when two transactions in a valid block modify the same object, the two modifications are elements of the same Abelian monoid. Deterministic operation follows from this commutativity.

Parallel Proposal and Execution The semantics of Groundhog are designed to enable effective parallelization of block proposal and execution while minimizing inter-thread synchronization overhead. Because each transaction in a block reads from a snapshot of state, and never reads data written by transactions in the same block, an implementation can largely avoid cross-thread synchronization. The only coordination required is for the maintenance of object constraints, which our implementation achieves using only hardware-level atomic memory instructions (§2.5.1). After iterating over all transactions in a block, Groundhog iterates (again, in parallel) over every modified object to produce the state snapshot for the next block.

To emphasize, Groundhog's parallelism is guaranteed for all replicas, both when proposing a block (as a leader) and when executing a proposal from another replica (as a follower). In fact, leaders and followers execute nearly the same code path. When the 2-phase locking algorithm of §2.5.1 identifies a conflict, a leader drops the offending transaction from the proposal it is assembling, while a follower raises an error and rolls back the block. There is never a fallback to sequential execution.

Semantic Flexibility Furthermore, Groundhog enables users to execute arbitrary code within a transaction, giving the system the flexibility to adapt to new use-cases and applications over time. The semantics of Groundhog are different from the strictly serializable semantics of many blockchains. However, as described in Fig. 2.1, we are able to implement many widely-used applications in these semantics, with minimal changes to the APIs these applications give to downstream smart contracts. Furthermore, these semantics are strictly more powerful than those based on message-passing actors (§2.4.6) used in some blockchains deployed in production today.

Application	Implementable	API Changes	Atomic Composability (§2.4.6)
Tokens (ERC20 Standard [301], §2.4.1)	✓	increaseAllowance, decreaseAllowance[151, 300]	✓
k-th Price Auction	✓	None	✓
Automated Market-Maker (Uniswap v2 [55], §2.4.6)	✓	None	✖
Money market (Compound v3 [37], §2.4.5)	✓	Separate Borrow and Supply Accounts	✓
Payment Channels [263]	✓	None	✓

Figure 2.1: A sampling of various applications implementable in Groundhog, and the changes required to implement them in Groundhog's commutative model.

Robustness to Malicious Contracts We emphasize that contracts in Groundhog can be untrusted, arbitrary scripts subject to no static analysis or manual review. Although of course smart contracts deployed on Groundhog may contain bugs, malicious or erroneous code cannot break down the operation of Groundhog. A transaction may consume excessive compute resources, but because each transaction executes in isolation against a static snapshot of state, a malicious transaction cannot manipulate the execution engine to cause, for example, worst-case performance of a concurrency control algorithm.

2.2.2 Design Motivation

Conceptually, in one block, Groundhog concurrently executes a collection of user-designed state machines. Instead of trying to accelerate these unknown state machines, Groundhog leaves it up to users to design their state machines to support concurrency. A state machine that relies on strictly sequential semantics can take out a lock on its state, but others might allow more fine-grained locking or may not even require locks. This isolates the performance (in terms of transaction throughput) of well-written state machines from that of incorrect, slow, or malicious state machines.

Our claim is not that every state machine can be reimagined as one that admits arbitrary commutative semantics. Rather, we argue that many important financial applications can be implemented to support at least some level of concurrent execution. And §2.4.6 gives a worst-case fallback option for contracts that need to interact with other contracts but require sequential semantics (such as constant function market makers, §2.4.6).

2.3 Commutative Semantics

The commutative semantics of Groundhog rely on two design features; first, all transactions in a block read from the same snapshot of state, and not the output of other transactions in a block. And second, in order to develop useful state machines, Groundhog carefully manages the typed output of transactions to resolve what would be write conflicts in other systems.

To implement these semantics, our Groundhog implementation provides to the WebAssembly runtime a set of typed methods to access persistent storage. It additionally provides a basic set of operations common to many blockchain contexts, such as $get\_block\_number()$, $get\_caller\_address()$, and $get\_self\_address()$.

2.3.1 Snapshot Reads

Groundhog provides a different notion of consistency than is typically used in distributed databases. Before processing a block of transactions (or assembling a new block to propose), Groundhog computes a snapshot of system state. Whenever a transaction queries for some address in the currently executing contract's local storage, the transaction reads from this snapshot, and no modifications made by other transactions in that block are visible.

These semantics may be unusual in existing systems, but they mirror the way in which users interact with blockchains today. Unless a user has a privileged relationship with a a block proposer, the user, when deciding whether to send a transaction, is typically only able to see the state of the blockchain at the close of the most recent block. Any transaction analysis or preflight simulation done by the user, therefore, will execute against this snapshot, without seeing any modifications made by other transactions in the block. The potential for differences between a user's snapshot view of state and sequential semantics has caused security challenges in real-world systems [300].

However, these semantics introduce the possibility of non-serializable execution traces. If a contract needs stronger serializability guarantees, Groundhog places the responsibility for ensuring serializability onto the contract. Contracts must take out locks on their internal state, if such locks are needed for the application. For example, these semantics admit write-skew anomalies, where one transaction might read two variables X and Y, and then write to Y, while another transaction in the same block reads the same two variables and writes to X. If such behavior is incompatible with the contract's higher-level application requirements, then the transactions should acquire a lock on X and Y (so only one could be included in a block).

Unlike designs based on eventual consistency and conflict-free replicated data types (CRDTs) [286], Groundhog's data structures need not support delayed updates from transactions arbitrarily far in the past, but rather, only concurrent updates from transactions in the same block. This property is crucial to implement, for example, asset transfers between account balances. CRDTs, for example, cannot maintain an integer counter that enforces a nonnegativity constraint and allows subtraction.

However, because a transaction in block k is guaranteed to see all of the outputs of transactions that were in block $k-1$ and none from transactions in block $k+1$, Groundhog's runtime can implement this primitive (§2.3.2) by looking only at a finite set of transactions. Furthermore, state machines can synchronize with each other at block boundaries (as in, e.g., §2.4.6), without requiring Groundhog to track complicated dependency chains between transactions.

2.3.2 Typed Storage Objects

Objects in Groundhog have distinct types, and are modified in structured patterns. This type information enables Groundhog to deterministically resolve concurrent writes to the same object in a way that is semantically useful, and to enforce constraints on object state.

Objects are created lazily with a type-specific default value when a transaction attempts to write to a nonexistent object. Two transactions conflict if they attempt to create objects of different types at the same location.

Transactions can delete keys, and two deletions on the same key do not conflict. As with writes, a deletion by some transaction is not made visible to other transactions in the same block, but rather deferred to the end of block processing, after all Abelian operations have been performed.

Groundhog provides a $\mathit{has\_key}()$ method if a transaction must know whether a particular object exists.

Byte Strings

Most applications need to store some amount of configuration data, be it public keys, expiration times, or contract addresses. Two transactions irresolvably conflict if they write different values to the same object. A write from one transaction overwrites an earlier write from that same transaction. The default value is an empty bytestring.

This is not an effective way to move money. If a user's balance is stored as a bytestring, then every access would need an exclusive lock (§2.4.2) so as to prevent the types of serialization anomalies mentioned in §2.3.1 (which could, for example, let a user spend money twice).

Nonnegative Integers

Groundhog uses nonnegative integers to represent account balances. Contracts can call $set(int64)$, $get() \rightarrow int64$, and $add(int64)$ methods on these values, and the Groundhog runtime ensures that the value of these integers does not drop below 0. This constraint would be impossible for a single transaction to enforce in the snapshot-read model. Integers implicitly created by a call to $add(\cdot)$ use a default value of 0.

A transaction that calls $set(x_1)$ conflicts with a transaction that calls $set(x_2)$ if $x_1 \neq x_2$, but two calls to $set(\cdot)$ of the same value do not conflict. If one transaction calls $set()$ multiple times, the transaction must input the same value to each $set()$ call. A collection of $add(\cdot)$ calls is invalid if they would drive the value of a nonnegative integer below 0.

This primitive allows Groundhog to efficiently implement digital money. The usage pattern for this primitive is to $set()$ the integer (an account balance) to its value in the prior snapshot, and then add or subtract from that value. A payment transaction is thus akin to calling $add(x)$ and $add(-x)$ on different balances—checking for an overdraft on an account is left to the Groundhog runtime (e.g., as in §2.4.1).

However, other usage patterns might use different values of $set()$; for example, a semaphore (§2.4.2) can be implemented as a method that always calls $set(1)$ and then $add(-1)$. Our implementation of a lending protocol (§2.4.5) $set()$s the value of an integer to an account's interest-adjusted borrowing limit. In fact, contracts can use this primitive to express to Groundhog arbitrary linear constraints on their state variables. Two linear constraints can enforce a requirement that an integer lie between 0 and some upper bound.

Additionally, we allow the input to $set(\cdot)$ to be negative; in this case, $add(\cdot)$ is invalid on any negative input.

Our implementation uses 64-bit signed integers, and caps the value of each integer accordingly (even if a set of transactions would have driven the value above `INT64_MAX`).

In a valid block, multiple transactions may $set()$ an integer to the same value, and then $add()$s to it. Therefore, when many transactions modify the same integer, the result, after applying all transactions, is equivalent to one $set()$ call followed by all $add()$ calls. To be precise, the value of an integer after a set of transactions that collectively call $\{set(X), ..., set(X), add(Y_1), ..., add(Y_k)\}$ is $\min(X + \sum_i Y_i,$ `INT64_MAX`$)$.

A contract implementing money might use a two-sided constraint for each balance (to prevent accidental loss of funds), or might limit the total supply of money to $INT64\_MAX$ (making overflow checks redundant). The choice is up to the implementer; Groundhog just supplies the minimal linear constraint primitive.

Ordered Sets

Any digital currency system needs a mechanism to prevent transaction replay; that is, users should be able to guarantee that each transaction executes at most once. Most blockchains use sequence numbers [23, 308] or ensure that each "coin" is spent only once [249, 36].

Contracts running in Groundhog can implement a replay prevention mechanism using "ordered sets". Each element is a pair $(tag, hash)$, where the tag is a 64-bit integer, and the hash is a unique 256-bit string. A transaction can $insert()$ an element (t, h). An $insert(\cdot)$ call fails if there is already some pair (t', h) in the set and conflicts with another transaction inserting (t', h) in the same block. The default value is the empty set. One could implement an idempotent set abstraction, allowing multiple insertions of the same value, but we did not need this abstraction.

To bound resource usage (and encourage garbage collection), we implement a maximum size on each set (specifically, each set starts with a default bound of 64, which can be increased by another method up to a maximum of 65535). Contracts can garbage collect set contents by $clear(\cdot)$ing a set of all entries, or clearing it of all entries below a specified tag value (semantically, Groundhog processes all insertion calls and then applies all $clear(\cdot)$ calls;

These sets can implement not just a replay prevention mechanism (§2.4.3), but also k-th price auctions (§2.4.4) and transaction sequencers (§2.4.6) in a message-passing "Actor" model. Objects in a set are sorted by the tag (with ties broken by the hash), which means that contracts can place their own notion of ordering on set elements. a replay prevention mechanism can order a set of transaction hashes by expiration time, and an auction can order a set of bids by their offered prices.

Note that it suffices, albeit complicating garbage collection, to represent sets of arbitrary data items with sets of 32-byte strings, by inserting to the set keys to persistent storage.

2.4 Evaluation: Semantic Flexibility

We demonstrate the flexibility Groundhog's semantics by discussing how we implemented a variety of applications commonly used on existing blockchains.

2.4.1 Tokens

Groundhog's semantics implement a contract that semantically represents a fungible token. In fact, Groundhog can implement the widely-used ERC20 standard token API [301] (although we have not implemented all of the possible extensions). The core logic is nearly identical to the token transfer logic used in existing blockchains (such as [135]).

Token balances are stored as nonnegative integers (§2.3.2). The primary difference between an implementation in Groundhog and traditional blockchains lies in the authorization framework. In the ERC20 standard, one contract can authorize another to transfer a bounded number of tokens on its behalf. Groundhog adjusts allowances by adding and subtracting to them, rather than setting them to particular values (as in the standard).

This API both enables an efficient implementation in Groundhog's commutative semantics, and avoids a double-spend attack present in existing blockchains [300]. Under sequential semantics, if one contract consumes an allowance immediately before a user changes the authorized amount, the contract can immediately withdraw the entirety of the newly authorized amount, effectively using more of the user's balance than the user intended to authorize. A widely used open-souce implementation of the standard [151] recommends this API change.

A minimal token implementation suffices itself to enable Groundhog to implement payment channels [263], atomic token swaps, and a subclass of decentralized exchanges, such as 0x [304], that perform offer matching off-chain. We have implemented a sample of these applications.

Empirical analysis of Ethereum's history [174] shows that the majority of conflicts between transactions that (under sequential semantics) prevent parallel execution result from integer counters and token balances (which are counters with a linear constraint). In other words, this historical analysis suggests that a vast majority of functionality could be implemented with just the nonnegative integer primitive (§2.3.2) and a set of exclusive locks (§2.4.2).

2.4.2 Locks

Applications may wish to ensure that an action is only performed once, or that the data that a transaction reads is not concurrently modified by another transaction (avoiding, e.g., the serialization anomaly outlined in §2.3.1).

The nonnegative integer *set* and *add* methods implement such a lock. A contract can dedicate a storage address to a nonnegative integer and acquire a lock by calling $set(1); add(-1)$ on that address. Recall from §2.3.2 that, conceptually, all of the *set* calls are applied before the *add* calls. Hence, a transaction that executes these operations can succeed once and only once in a batch of transactions.

This interface implements shared read-write locks as well. Exclusive access translates to a $set(1); add(-1)$ call, while shared access translates to a $set(0)$ call. That said, Groundhog does not have a mechanism for prioritizing which transactions succeed in the case of contention on a lock. A rudimentary solution would be for readers and writers to divide access between even and odd block numbers.

2.4.3 Replay Prevention

A transaction can ensure that it only executes once by inserting a hash of itself into one of the ordered sets of §2.3.2. This would cause the transaction to conflict with any concurrent or future invocation of itself. We imagine that most user transactions would first call into a "wallet" contract implements such a mechanism (as well as performing authorization checks, such as signature validation).

In this context, the tag on the ordered set acts as a garbage collection measure. It would be impractical for a ledger to store indefinitely (outside of high-latency archival storage) the hash of every transaction in its history; most blockchains implement or propose some way to reclaim stale data [87, 20, 109].

Therefore, a replay mechanism can include on each transaction an expiration time (batch number). The mechanism can check whether the transaction has expired, and (if not) insert the transaction's hash into a set, using as the tag the expiration batch number. Clearing the set of all entries with tags less than the current batch number thus garbage collects entries no longer necessary for replay prevention.

2.4.4 Auctions

Auctions form another core component of many financial applications. Some protocols, such as MakerDAO (the service behind the Dai stablecoin [14]), use auctions to liquidate collateral when a borrower's collateral to debt ratio falls. Many services issue indivisible assets (such as NFTs [160]) using competitive auctions. Contention of many transactions on a small set of auctions can become so fierce as to take down entire blockchains [280].

Put simply, the input to an auction is a set of bids, and the output is the winner (or winners) and a winning price. Typically, the highest bidder pays the bid offered by the second-highest bidder.

We implement such an auction using the ordered sets of §2.3.2. Users submit records of bids (an offered price, and the user's identity), which are then hashed and inserted into a set. The tag associated with the bid is the offered price. After the auction closes (either by an external action or after a certain time passes) the bids in the set will be in an order sorted by their price—so a contract can easily compute the second highest bid. Users send the auction contract capital to cover their bids when creating a bid, and can refund this capital after the auction ends. To prevent a bid from being refunded twice, users insert the hash of their bid into a second set after activating a refund.

A second-price auction need only keep track of the top two bids. Low bids (that might otherwise fill a set to the maximum limit) can be refunded and cleared from the sets (both the bid set and the refund replay-prevention set) when necessary. When receiving a new bid, the contract should check to ensure that the new bid offers a price at least as high as the current second-highest bid. To avoid filling up the ordered set, the contract could also require each bidder to refund and delete a superseded bid, though we have not yet implemented such a feature.

2.4.5 Money Market

We additionally implemented a lending and borrowing protocol. We specifically implemented version 3 (the most recent version) of the Compound protocol [37, 221], which currently manages several billion dollars worth of assets [44]. Other money market protocols operate on similar principles. In this system, users can supply or borrow a "base" asset (in practice, a USD-pegged token). Lenders and borrows receive and pay interest, respectively.

Interest is implemented through shared compound interest index variables. Base asset amounts are stored relative to amounts hypothetically deposited at the beginning of the protocol and converted to present value using these indices (so updating the indices implicitly updates every user's present balance). Borrowing and lending occur at different rates, and maintain separate indices.

In the sequential implementation, each transaction that touches the protocol updates these indices, based on the time elapsed since the indices were last updated. Because time advances once per block in a blockchain, these indices are effectively updated once per block. This type of update is implementable naturally in Groundhog's semantics, as two writes of the same value to a bytestring do not conflict.

In Compound, borrowing is overcollateralized by deposits of other assets. Users are not allowed to borrow more of the base asset or withdraw collateral if that would cause their existing borrowing to be undercollateralized. Groundhog implements this via a linear constraint on each user's state (using a nonnegative integer primitive). As collateral prices may fluctuate and user balances change each block (due to interest), the value of this constraint may change from block to block. As such, every transaction on the protocol recomputes the maximum remaining amount a user can borrow in that block, and sets the value of the constraint (via the $set(\cdot)$ method) to this value, before adding and subtracting to this value (as the user performs actions). Two transactions in the same block from the same user will compute the same initial constraint, and therefore this constraint does not cause a conflict.

Compound computes variable interest rates based on the relative amounts of the base asset borrowed from and supplied to the contract. Our implementation uses contract-wide integer counters to compute these values, but to update these counters, each transaction must know whether a transfer of the base asset out of the account is borrowing more or supplying less. The sequential implementation stores a user's base asset balance as a single, signed integer. However, our implementation must split a user's base asset balance into two nonnegative integer counters, one for borrowing and one for supplying. Transactions then specify whether a transfer is to reduce supply or to borrow more (or some combination), instead of having these amounts be computed automatically. This could result in a user simultaneously borrowing and supplying, so we also add a method to cancel out this situation.

2.4.6 Sequencers and the Actor Model

The ordered set mechanism of §2.3.2 can add back sequential (strictly serializable) semantics to contracts in which noncommutativity is inherent. For example, in each batch, a contract can create a new set, and transactions can submit an action (i.e. a function call) to this set. Then, in a subsequent batch, one transaction can load all of the actions from the set of a previous batch (e.g., loading batches in sequential order), and apply all of the actions in sequence in that transaction. Certain decentralized exchanges, such as Serum [24] and SundaeSwap [292] use precisely this type of mechanism already, even when the underlying blockchain guarantees strictly serializable semantics. The implementation allocates one ordered set per block, in which events are accumulated. Then, at a later block, anyone can iterate over and apply all events accumulatd in the earlier block.Our implementation sorts accumulated events by a fee bid, which is collected and paid out to whoever executes the events.

Observe that if access to every contract were gated by such a sequencer, then the result would be a world where all contracts would only interact via asynchronous message-passing (the Actor Model [195]). This is precisely the semantics of contracts implemented in CosmWasm [38] and the Near blockchain's [15] runtime system [40].

In other words, Groundhog's semantics are at least as expressive as the semantics used in production today by two of top 50 (by market capitalization [130]) blockchains. However, message passing semantics do not allow atomic transactions between contracts. Groundhog, therefore, is strictly more powerful than this message-passing model because it enables contracts that do not need sequencing to call one another directly—which is especially important for fundamental, commonly-used contracts such as token contracts.

Example: Constant Function Market Makers

We used the sequencer paradigm to implement a Constant Product Market Maker [55, 65]. Users submit operations (make a trade, deposit liquidity, or withdraw liquidity). All of the operations submitted in one Groundhog block are gathered into the same ordered set. Our implementation requires users to pay fees on their transactions. Fees are collected by whoever sends the transaction that executes the set of operations, as in [24]. The fee paid is used as the tag on the set (thereby sorting operations by fee).

The main Groundhog introduces is that the runtime cannot reject individual operations on the exchange as they execute in one transaction. If any operation causes the overall transaction to conflict with another transaction (e.g., by overdrafting an account), then Groundhog will reject the entire transaction and not execute any operations in the set.

As such, a good implementation would perform any changes to persistent storage that could cause a conflict (such as withdrawing money from an account) in the transaction that submits an operation to a set, not during the transaction that applies a set of operations. This would mean withdrawing sufficient assets to cover a trade when sending an operation to the sequencer, not when executing the event. And the transaction that executes a set of operations should be the only one logically allowed to disburse funds from the contract.

2.5 Implementation

Our implementation of Groundhog consists of approximately 22k lines of C++. Our contract SDK is currently written in C++, but Groundhog itself is agnostic to the choice of source language. Contracts are compiled to WebAssembly and then executed using the Wasm3 interpreter [27]. Before executing a contract, Groundhog instruments the WebAssembly module with execution duration metering ("gas metering") via [43].

Groundhog and our example contracts are available open-source at `https://github.com/scslab/smart-contract-scalability`.

2.5.1 Block Proposal via 2-Phase Locking

When assembling a block of transactions or executing a proposed block from another replica, Groundhog first executes each transaction individually, which translates each transaction from a WebAssembly function call into a list of changes to be applied to persistent storage. It then uses a process akin to 2-phase locking to decide whether a transaction conflicts with another transaction already added to the proposed block (or with another transaction in the block, when executing another replica's proposal).

The simplest manner for performing these checks would be, for each transaction, to iterate over the transaction's changes and acquire a lock on each relevant object in persistent storage. Then, with the locks acquired, a thread could check whether each change to an object is consistent with the object's constraint, and then either commit or unwind all changes before releasing the locks.

Of course, a design based on exclusive locks on shared data does not scale when many transactions modify a single object. Our implementation lets threads reserve the right to later perform modifications to state using only hardware atomics. A reservation to write a bytestring at some address can be granted, for example, if there is no reservation (committed or otherwise) to write a different bytestring at that address, and this can be implemented using an atomic pointer swap, an atomic counter, and unique tags.

Similarly, a reservation for a $set(x_1)$ call fails if another transaction has reserved $set(x_2)$ with $x_1 \neq x_2$ on that nonnegative integer. Reservations for $add(y)$ always succeed for $y \geq 0$ (overflows are capped at $INT64\_MAX$), but fail when $y < 0$ if the total amount reserved for subtraction would exceed the base value of the counter. Internally, Groundhog compiles all of the calls to $set()$ and $add()$ from one transaction on one integer into a single "$set\_add(x, y)$" operation—to know whether the initial value of an object is high enough to subtract y from and remain nonnegative, Groundhog must know what the initial value is.

This system, combined with the commutative semantics of §2.2 , lets a block proposer guarantee that after applying all transactions in a proposed block, each storage object is in a valid state. In fact, our implementation guarantees further that every subset of a valid block is itself also a valid block.

2.5.2 Additional Details

Our implementation stores persistent state in a Merkle-Patricia trie; this is required for an implementation to produce short proofs of state or of a transaction's execution. After executing a batch of transactions, Groundhog must iterate over all modified storage objects to compute a new state snapshot for the next batch. It does this by first building another Merkle-Patricia trie that logs only which storage locations were modified in the batch, and uses this record to efficiently iterate over the first (possibly much larger) trie. Groundhog also builds a trie logging all of the transactions in a batch. Finally, Groundhog recomputes the root hashes of each of these tries.

Our goal is to implement the features important to a real-world deployment that require direct integration with Groundhog. A deployment of a digital currency would include Groundhog as one part of a much larger system. Our implementation stores all data in memory, but a deployment would log transactions to persistent storage. Groundhog builds a trie capturing all of the modified storage locations to enable an efficient implementation to identify all locations that must be rewritten to persistent storage.

Our implementation includes transaction runtime metering but not a default unit of money. A deployment could implement a fee token using a standard token contract (§2.4.1) deployed at a hard-coded location.

Batching for Qualitative Simplification We observe that our batch execution model can lead to qualitatively simpler implementations than we otherwise would need. For example, we avoid the multi-version data structures common in other forms of concurrency control. Garbage collection can be deferred to the end of a batch, at which point each thread can know that no other thread is using a reference to any of its locally-accumulated garbage memory.

Additionally, we implement our set objects (§2.3.2) using atomic hash sets. The sets themselves contain only 4-byte pointers (offsets) into threadlocal static buffers. Allocation of a hash set entry simply increments a threadlocal counter, and the set implementation needs no locks, only hardware atomic memory operations. After a batch of transactions, the contents of the atomic hashset are moved to a snapshot and the hashset is cleared, so our atomic hashset implementation needs only to provide a limited interface (insert, erase, and a single dump at the end of the block).

2.6 Evaluation: Scalability

We compare the throughput and scalabiltity of Groundhog against Block-STM [179], an approach based on optimistic concurrency control for deterministic execution of batches of transactions on a replicated state machine. Like Groundhog, Block-STM executes transactions that contain untrusted, arbitrary user-submitted code and targets financial applications. It is in production use by the Aptos blockchain [288].

Our experiments in this section run on one c6a.metal instance in an Amazon Web Services datacenter. The system has two 48-core AMD EPYC 7R13 processors and 384 GB of memory. We ran our experiments with hyperthreading disabled.

Payments Workload To control for transaction complexity, the performance measurements of this section use payment transactions, modeled on the “Aptos p2p” payment transactions of [179]. These transactions read from 8 memory locations and write to 5 locations, while the implementation of this functionality in Groundhog uses 3 adjustments to nonnegative integers (using the ERC20 token implementation of §2.4.1), two reads of configuration data, one set insertion, and one set clear (for the replay prevention mechanism of §2.4.3). This roughly translates to reading 6 memory locations and writing to 4. Each transaction verifies an authorization signature. Of course, in any transaction execution engine, transactions that require more computation would run more slowly and reduce throughput numbers accordingly.

Each transaction is a payment between two accounts, chosen uniformly at random. Varying the number of accounts therefore varies the level of contention between transactions.

Varying Contention Fig. 2.2 plots the end to end transaction throughput rate of Groundhog on batches of payment transactions. The measurements are averages over 10 trials, after 5 warmup rounds. These experiments vary the number of accounts and the size of each batch of transactions.

There is a certain amount of work that must be performed once per batch; larger batch sizes amortize this constant (or less effectively parallelizable) work over more transactions. The replay prevention sets in this experiment expanded to their maximum capacity (65535). When serialized, each transaction is 196 bytes in size, so $100,000$ transactions per second would need a network capable at minimum of handling at least 20 MB/s of sustained traffic.

What makes Groundhog unique is that the level of contention between transactions does not significantly affect throughput. When every payment transaction is between the same two accounts, then every transaction modifies the same memory locations. Under any approach using sequential semantics, all transactions in this scenario would conflict and execution would be inherently serialized. However, Groundhog achieves the same throughput in this high-contention setting as in a very low contention setting.

As another comparison, SPEEDEX [269] achieves a similar level of scalability on payment transactions, using a design also based on batched execution of commutative transactions. However, SPEEDEX only supports a small set of fixed types of transactions, specialized for one application, whereas transactions in Groundhog can execute arbitrary, untrusted computation.

By contrast, Fig. 2.3 plots the throughput of Block-STM on the same hardware and transaction distribution patterns. This design is based on optimistic concurrency control, which must fall back to nearly sequential execution under high contention (while still paying for concurrency control overhead).

In the real world, it may be unlikely that two individuals would send bursts of thousands of transactions between each other; however, it is quite likely that single accounts may send or receive bursts of payment traffic. Large e-commerce websites or the Internal Revenue Service, for example, might need to occasionally receive a high volume of payments, and accounting services might need to send salary payments to a large number of employees in a short period of time. Groundhog's commutative semantics allow institutions to build scalability into their payments infrastructure and simultaneously guarantee that other users are not affected by contented data elsewhere in the system.

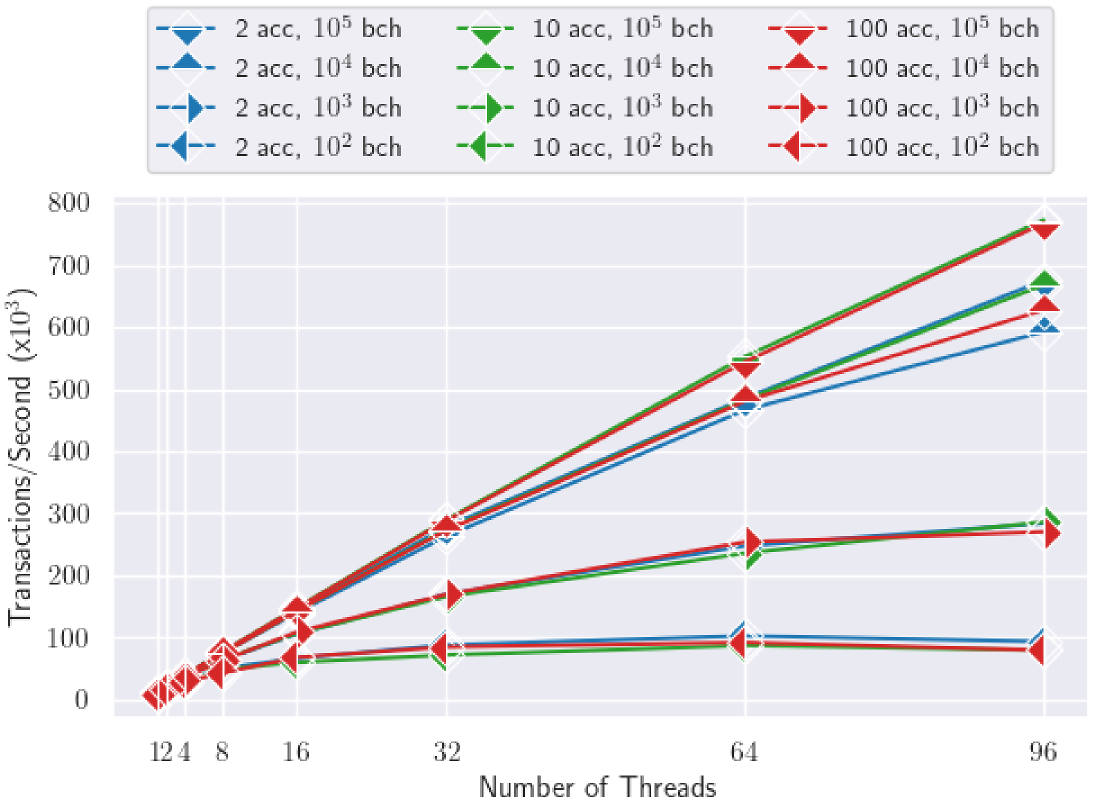


Figure 2.2: Transactions per second on Groundhog, varying the number of accounts ("acc") and batch size ("bch").

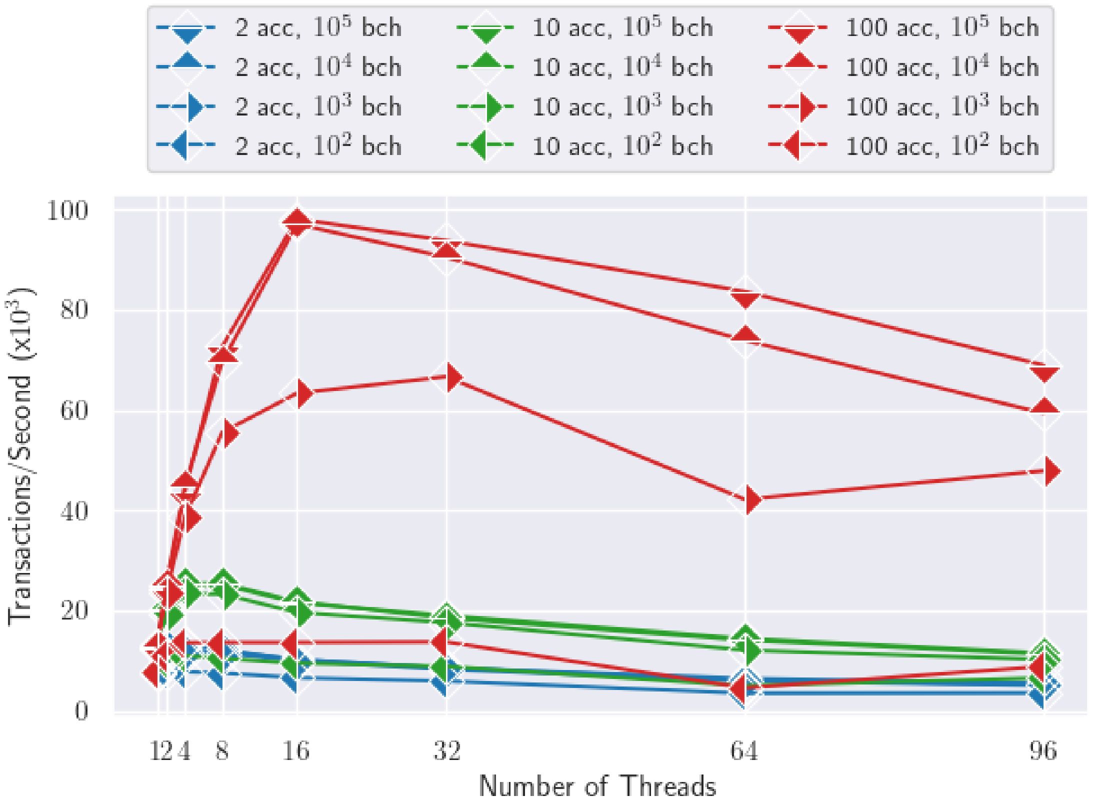


Figure 2.3: Transactions per second on Block-STM [179], varying the number of accounts ("acc") and batch size ("bch").

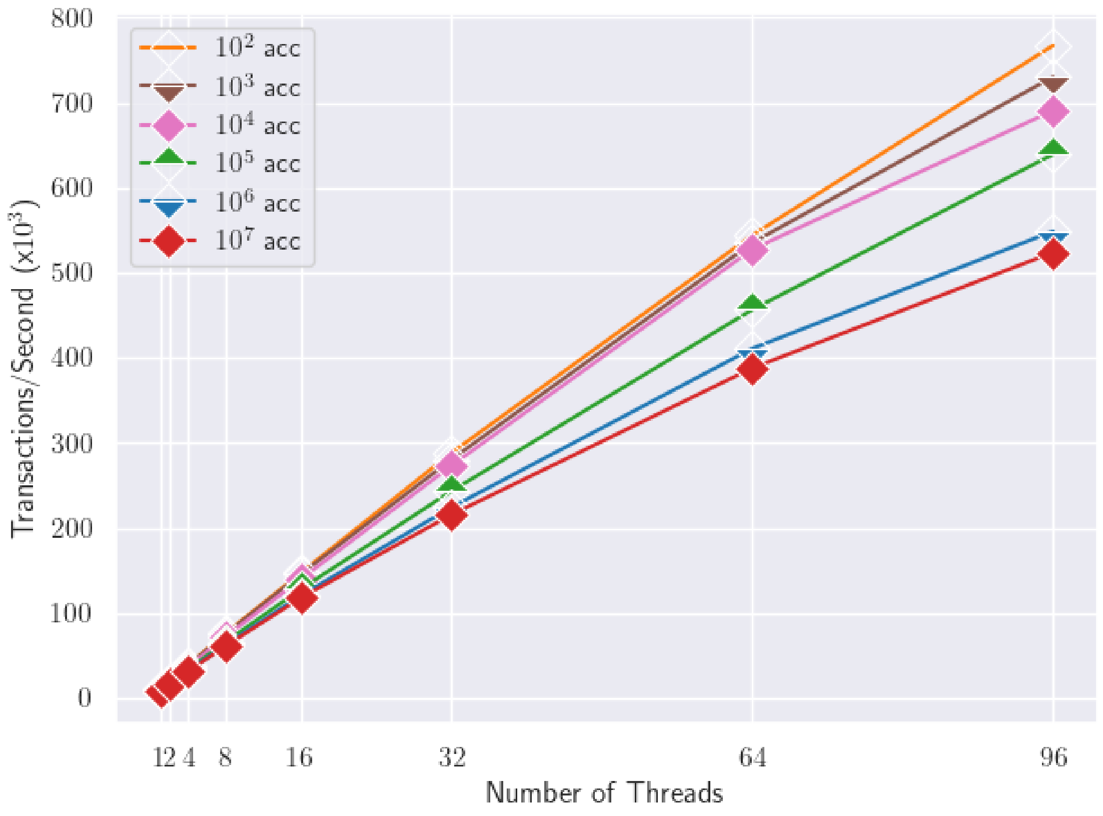


Figure 2.4: Transactions per second on Groundhog with batches of size 100,000, varying the number of accounts.

Scalability to Many Accounts The cost of access to a user's account data increases as the number of users increases. Fig. 2.4 plots the throughput of Groundhog as the number of accounts increases, with a fixed batch size of $100,000$.

The replay prevention sets (§2.4.3) are at their default capacity (64), except when the number of accounts is 1,000 or smaller, when we again expand them to their maximum capacity (65535). Our lockfree hash set implementation (§2.5) uses 4 bytes for every slot of capacity (with a preallocated buffer), and our system does not have enough memory to support 10 million accounts with maximum capacity sets (this would require over 2 terabytes of memory). However, when the number of accounts is small, each account must be able to send a large number of transactions in a batch (to reach a batch size of $100,000$).

As expected, the throughput of Groundhog decreases as the number of users increases, largely due to the increased lookup time per account. The single-threaded throughput with 10 million accounts, for example, is 88% of the throughput with 10 thousand accounts. More accounts also

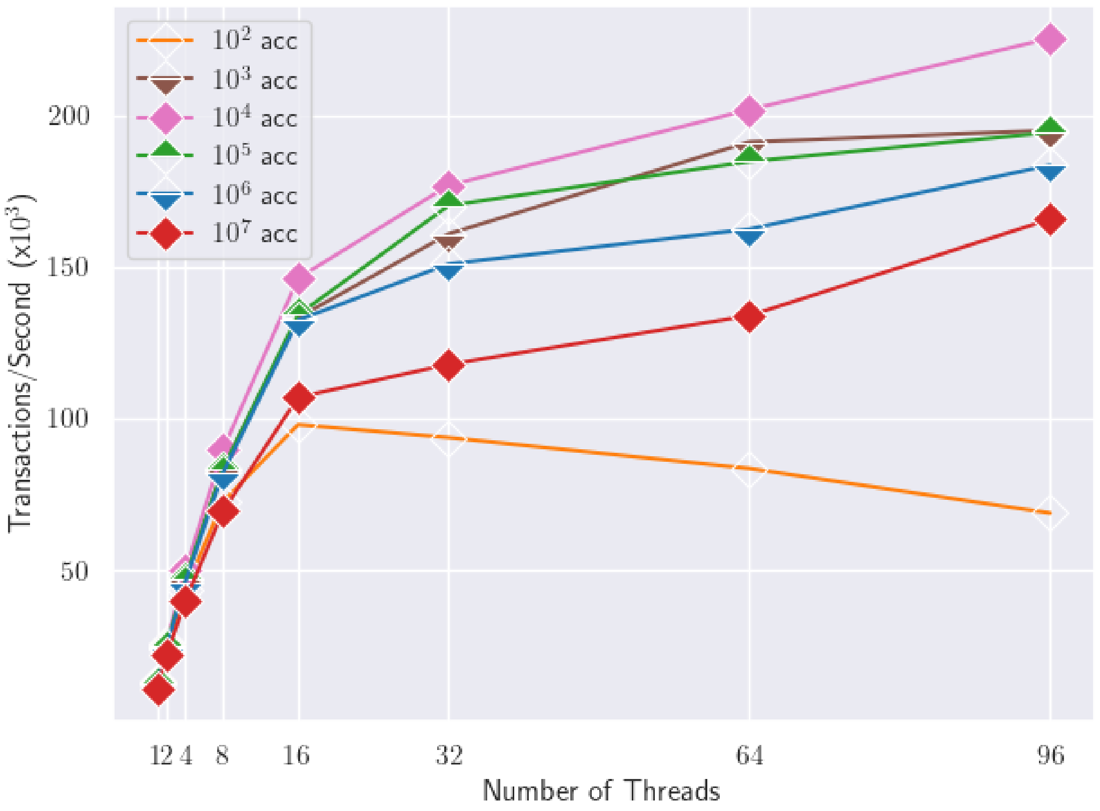


Figure 2.5: Transactions per second on Block-STM with batches of size 100,000, varying the number of accounts.

slows down the work that is done in each block after iterating over the transactions, slightly reducing scalability. The throughput with 96 threads and 10 million accounts is, for example, 76% of that with 10 thousand accounts (a 75× speedup vs. a 64× speedup over the single-threaded benchmark, respectively).

We do not try to optimize Groundhog for the precise details of our machine's NUMA architecture; we suspect that some of the scalability decrease comes from increased variability of memory access times when all cores are active.

As a comparison, Fig. 2.5 plots the throughput of Block-STM on this workload. Like Groundhog, Block-STM's throughput falls as the number of accounts increases. However, note that scalability is quite limited, even in the extreme case where there are millions of accounts and there is virtually no contention between transactions. The overhead of concurrency control appears to become the dominant runtime factor, even when there are no concurrency conflicts.

Conclusions More important than any topline number is the scalability of Groundhog. As discussed in §2.5, our implementation includes what we believe to be a minimal complete set of features. However, any deployed system may want additional features, each of which adds overhead to each transaction. For example, transactions might require multiple authorization signatures, or an implementation might choose a slower but simpler (and easier to audit) implementation of the WebAssembly sandbox.

Sequential semantics inherently force a deployment to choose between more features and higher throughput. SCS adds a dimension of cost to this tradeoff. A deployment can compensate for additional features (and any resulting throughput drop) by simply adding additional compute hardware. It may be unlikely that any near to medium term deployment of a digital currency would need multiple hundreds of thousands of transactions per second, but this scalability enables a deployment to smoothly compensate for future feature demands and increased workload.

These measurements only count as throughput the rate of successful transactions. If many transactions sent to Groundhog are invalid for some reason, then when proposing a batch, a replica would execute these transactions but not include them in a proposal. The measured throughput would fall accordingly, but the scalability properties of Groundhog would be largely unaffected (in fact, transactions that are not included in a proposal do not influence the amortized per-batch overhead). To put these numbers in context, it may be quite unlikely that any near to medium term deployment of a digital currency would need multiple hundreds of thousands of transactions per second. [1]

2.7 Extension: Incomplete External Calls

Groundhog's commutative semantics enable behavior not directly implementable in a sequentially executing state machine. When assembling a proposed block, Groundhog can pause the execution and reallocate resources to a different transaction, without disrupting the execution of any other transaction. This enables transactions in Groundhog to call into external, possibly incomplete computations. If an external call fails to terminate (within a time bound), Groundhog can simply drop the transaction, without affecting the operation of any other transaction or wasting any compute resources (besides those for the dropped transaction).

Replication requires that the results of external calls be self-authenticating (e.g., signed by a relevant authority). A block proposer must include these results in its block proposal. Access to external information is an important business in existing blockchains [100], but suffers from high latency, as queries can only return results in subsequent blocks. These calls in Groundhog let contracts directly access external information; for example, a bank might hold personal information privately and authorize transfers based on queries to this data without revealing the private information.

[1] If you're reading this far into this thesis, you must be interested in the topic and I'd love to chat with you! Feel free to send me an email, and I'll buy you a coffee if you mention you found this footnote.

Atomic Swaps External calls can enable efficient off-chain market-making, addressing the so-called "free option" problem of existing decentralized exchanges that execute atomic swaps [189, 262]. In an atomic swap, two users agree to a trade between two assets at some exchange rate. Implementing this requires making two asset transfers in one transaction, and requires authorization from both users. Typically, one user signs the transaction first, and then sends the transaction to the other, who signs it and submits it to the blockchain.

However, upon receipt of the first user's signature, the second user has the option of dropping the transaction. To minimize swap failures due to network delay or blockchain congesion, swap transactions often include a relatively long expiration time. The optionality benefits the second user at the expense of the first [189].

By contrast, in Groundhog, if a swap is initiated within a transaction via a call to an external service (i.e., a request for quote API), both swap participants know that if the swap settles, then it will settle within the same block—within the smallest discrete amount of time that a blockchain can measure. The Groundhog replica has the option to cancel the swap, but neither of the swap participants do.

2.8 Design Limitations

Development Difficulty The execution semantics of Groundhog may make developing applications more difficult than in a sequentially executing system. Anecdotally, we found it helpful to conceptualize a contract's API as a set of actions that a caller might perform, subject to a set of constraints. Our conjecture is not all but a majority of important applications can be built in a manner that admits some commutativity. Ramseyer et al. [269] additionally find that redesigning an API to add commutativity can introduce useful economic properties.

However, as discussed in §2.4.6, production blockchains today [38, 15] use semantics, based on asynchronous message passing, that are strictly weaker than those of Groundhog. This gives a natural baseline implementation for any contract used in those systems. We do not implement it here, but message-passing designs may wish to programmatically schedule a transaction to execute in a future block.

Leader-based Design Groundhog's commutative semantics eliminate the ability of one replica to manipulate the ordering of transactions for its own profit within a batch, but a malicious batch proposer might easily censor or delay transactions. Censorship and delay resistance are properties of a consensus protocol, not the execution engine. We envision Groundhog in an environment with large-scale, regulated financial institutions operating replicas, where regulation ensures non-malicious replica operation in most cases, but Groundhog's leader-based proposal process complicates an integration with leaderless consensus protocols [144, 206] that may be more censorship-resistant.

The challenge is that Groundhog cannot maintain object constraints if transactions in a batch conflict. If Groundhog is given a batch that may cause a constraint violation it must first remove sufficient transactions to prevent any violations. We do not implement it here, but Groundhog could add a generalization of the deterministic filtering mechanism of [269].

Specifically, Groundhog could execute all transactions in a block, and identify any objects on which transactions conflict or that are put in a constraint-violating state. Groundhog can then compute, for each object, a "conflict set" of conflicting transactions. For example, if several transactions write different values to a bytestring, then all of these transactions are in the conflict set. If a nonnegative integer's constraint is broken, then all transactions that subtract from its value form the conflict set. Removing transactions cannot cause additional conflicts in Groundhog, so removing the union of all conflict sets leaves a block with no constraint violations.

2.9 Related Work

2.9.1 Semantic Changes

Our approach is modeled on the Scalable Commutativity Rule of Clements et al. [125], who build scalability into an operating system by designing commutative system call semantics. CoWSwap [7, 12] and Penumbra [17] also implement specific state machines that execute transactions in unordered batches. Our design of SPEEDEX in chapter 3 uses a similar semantic model to operate a decentralized exchange that concurrently executes batches of unordered transactions.

Garamvölgyi et al. [174], resolve some write conflicts by adding a "commutative add" instruction. Their design, however, maintains sequential semantics, and does not support subtraction from a counter in a commutative manner. Studies of historical Ethereum data [278, 174] find that most conflicts come from token contracts and other integer counters.

Conflict-free Replicated Data Types [286] give commutative interfaces to data structures in a distributed environment, but guarantee that operations can be applied after any amount of delay. They cannot, therefore, implement an account balance that admits subtraction while ensuring nonnegativity.

2.9.2 Parallel Execution

Block-STM [179] optimistically executes transactions in parallel, re-executing in case of conflict. This meets our requirements for a digital currency engine, except that its throughput does not scale under limited contention. Chen et al. [120] speculatively execute Ethereum transactions, and Lin et al. [229] instrument Ethereum execution traces to reduce re-execution overhead after concurrency conflicts.

Eve [204] executes batches of transactions concurrently, but relies on an application-specific conflict checker to assemble batches of nonconflicting transactions. This checker requires advance knowledge of transaction read and write sets, and performance degrades substantially when this checker produces errors. Malicious users could attack any conflict checker with malicious code to introduce errors. By contrast, transactions in Groundhog execute arbitrary, untrusted code, and Groundhog needs no advance knowledge of read or write sets.

LiTM [309] takes a batch of transactions and executes on a single snapshot as many nonconflicting transactions as possible, then recomputes a new state snapshot and continues. However, throughput is limited by round frequency, because a data item can only be updated once per round, and Gelashvili et al. [179] found throughput scalability of LiTM to be limited under contention.

Strife [264] dynamically splits a batch of transactions into partitions where transactions in different partitions do not conflict, and executes these subsets in parallel. Bohm [164] deterministically executes batches of transactions using a multi-version data structure. Both approaches rely on advance knowledge of transaction read and write sets.

Basil [289] builds a key-value store in a context where replicas and users may be faulty, boosting performance by adapting optimistic techniques to this setting. However, the system relies on users to execute transactions, which means that it cannot ensure correct execution or prevent Byzantine clients from overwriting the data of correct users.

Other approaches compute nonconflicting transaction sets [89, 317, 196] or include conflict resolution information in block proposals [150, 71, 70]. Ruan et al. [277] reorder transactions to reduce conflict rates.

2.9.3 Ordering Semantics

Li et al. [226, 225] propose RedBlue consistency, in which a transaction can be broken into non-commutative (Red) and commutative (Blue) components. Red components must still be executed sequentially, limiting scalability, and Blue components may or may not be visible to Red ones, making system operation nondeterministic. PoR consistency [227] generalizes this notion to more fine-grained ordering requirements, but has the same challenges in our context. Systems like COPS [231] likewise relax consistency guarantees to increase performance but do not guarantee reproducible operation.

As discussed in §2.3.1, Groundhog deliberately builds no ordering between transactions in a block. Other decentralized systems build a “fair ordering” assuming a bounded fraction of malicious replicas [209, 322, 100, 267] or allow a replica to cryptographically commit to an ordering before revealing the contents of a transaction [126, 320, 282, 184].

2.9.4 Divided State

Several blockchain scaling approaches divide blockchain state into independently-running state machines (“shards”) [9, 321, 307, 291, 145] or side-chains (“Layer-2 networks”) [263, 203, 18, 10, 262, 11, 35, 183]. Each component can run independently, but these designs complicate any interaction across components and limit throughput between components.

Some designs operate each smart contract independently (in parallel), allowing communication between contracts only through asynchronous messages [38, 40, 285, 202]. This model complicates many common tasks, such as asset transfers (which would ideally be done atomically), as each call to a token contract is asynchronous.

Several systems, such as Bitcoin [249] and Project Hamilton [234], divide user balances into pieces, where each piece is an “Unspent Transaction Output” (UTXO). Use of UTXOs enables some parallelism (as in e.g. [313]), but complicates the user experience—each user must track a large set of UTXOs, not just one account balance—and complicates any abstraction built on top of them. Project Hamilton [234] furthermore observed that even committing to a total ordering between payment transactions became a throughput bottleneck. Groundhog, by contrast, gets its scalability without making this tradeoff.

2.10 Conclusion

We presented here Groundhog, our novel design for a smart contract execution engine. Groundhog is designed to execute blocks of transactions in parallel, but unlike prior work, Groundhog semantically places no ordering between transactions in a block.

Groundhog’s key insight is to design a set of commutative semantics that allow the Groundhog runtime to deterministically resolve concurrent writes to the same object. To support contexts where transactions may irresolvably conflict (such as when withdrawing money from an account), Groundhog lets contracts express constraints about their data objects. Groundhog maintains these constraints by using a 2-phase locking protocol to assemble and propose blocks that contain no conflicts.

These semantics enable Groundhog to execute transactions in parallel with minimal synchronization overhead between threads. Crucially for any real-world deployment of a digital currency system, this parallel execution enables Groundhog to scale its transaction throughput as it is given more compute resources. Furthermore, the commutativity ensures that the rate at which transactions modify the same objects (which would lead to conflicts in designs based on optimistic concurrency control) does not meaningfully affect throughput.

Our implementation of Groundhog achieves more tha half a million payments per second between 10M accounts, but achieves even more than that when transacting between only 2 acccounts.

Chapter 3

SPEEDEX: A Scalable, Parallelizable, and Economically Efficient Decentralized EXchange

3.1 Introduction

Digital currencies are moving closer to mainstream adoption. Examples include central bank digital currencies (CBDCs) such as China's DC/EP [256], commercial efforts [197, 169], and many decentralized-blockchain-based stablecoins such as Tether [293], Dai [14], and USDC [26]. These currencies vary wildly in terms of privacy, openness, smart contract support, performance, regulatory risk, solvency guarantees, compliance features, retail vs. wholesale suitability, and centralization of the underlying ledger. Because of these differences, and because financial stability demands different monetary policy in different countries, we cannot hope for a one-size-fits-all global digital currency. Instead, to realize the full potential of digital currencies (and digital assets in general), we need an ecosystem where many digital currencies can efficiently interoperate.

Effective interoperability requires an *exchange*: an efficient system for exchanging one digital asset for another. Users post offers to trade one asset for another on the exchange, and then the exchange matches mutually compatible offers together and transfers assets according to the offered terms. For example, one user might offer to trade 110 USD for 100 EUR, and might be matched against another user who previously offered to trade 100 EUR for 110 USD. A typical exchange maintains *orderbooks* of all of the open trade offers.

The ideal digital currency exchange should, at minimum,

- not give any central authority undue power over the global flow of money,
- operate transparently and auditably,
- give every user an equal level of access,
- enable efficient trading between every pair of currencies (make effective use of all available liquidity), and
- support arbitrarily high throughput, without charging significant fees to users.

Scalability is crucial for a piece of financial infrastructure that must last far into the future, as the number of individuals transacting internationally continues to grow. Furthermore, the above feature list is by no means complete; a deployment may want any number of additional features, such as persistent logging, simplified payment verification [249], or integrations with legacy systems, each of which slows down the system's performance. Scalability, viewed from another angle, enables the system to add features without decreasing overall transaction throughput (at the cost of additional compute hardware).

The gold standard for avoiding centralized control is a *decentralized exchange*, or DEX: a transparent exchange implemented as a deterministic replicated state machine maintained by many different parties. To prevent theft, a DEX requires all transactions to be digitally signed by the relevant asset holders. To prevent cheating, replicas organize history into an append-only blockchain. Replicas agree on blockchain state through a Byzantine-fault tolerant consensus protocol, typically some variant of asynchronous or eventually synchronous Byzantine agreement [115] for private blockchains or synchronous mining [249] for public ones.

Unfortunately, existing DEX designs cannot meet the last three desiderata.

Equality of Access In existing exchange designs, users with low-latency connections to an exchange server (centralized or not) can spy on trades incoming from other users and *front-run* these trades. For example, a front-runner might spy an incoming sell offer, and in response, send a trade that buys and immediately resells an asset at a higher price [97, 223]. In a blockchain, where a block of trades is either finalized entirely or not at all, this front-running can be made risk-free. More generally, some users form special connections with blockchain operators to gain preferential treatment for their transactions [143]. This special treatment typically takes the form of ordering transactions in a block in a favorable manner. The result is hundreds of millions of dollars siphoned away from users [265].

Effective Use of Liquidity Existing exchange designs are filled with arbitrage opportunities. A user trading from one currency A to another B might receive a better overall exchange rate by trading through an intermediate *reserve* currency C, such as USD. Users must typically choose a single (sequence of) intermediate asset(s), leaving behind arbitrage opportunities with other intermediate assets. This challenge is especially problematic in the blockchain space, where market liquidity is typically fragmented between multiple fiat-pegged tokens.

Computational Scalability DEX infrastructure must also be scalable. The ideal DEX needs to handle as many transactions per second as users around the globe want to send, without limiting transaction rates through high fees. Trading activity growth may outpace Moore's law, and should not be limited by the rate of increase of single-CPU-core performance. An ideal DEX should handle higher transaction rates simply by using more compute hardware.

Unfortunately, folk wisdom holds that DEXes cannot scale beyond a few thousand transactions per second. Naïve parallel execution would not be replicable across different blockchain nodes. This wisdom has led to many alternative blockchain scaling techniques, such as off-chain trade matching [304], automated market-makers [55], transaction rollup systems [21, 30], and sharded blockchains[9] or side-chains [262]. These approaches either trust a third party to ensure that orders are matched with the best available price, or sacrifice the ability to set traditional limit orders that only sell at or above a certain price (reducing market liquidity). Offchain rollup systems, sharded chains, and side-chains further fragment market liquidity, leading to cross-shard arbitrage and worse exchange rates for traders.

A challenge for on-chain limit-order DEXes is that the order of operations affects their results. Typically, a DEX matches each offer to the reciprocal offer with the best price: e.g., the first offer to buy 1 EUR might consume the only offer priced at 1.09 USD, leaving the second to pay 1.10 USD. Each trade is a read-modify-write operation on a shared orderbook data structure, so trades must be serialized. This serialization order must be deterministic in a replicated state machine, but naïve parallel execution would make the order of transactions dependent on non-deterministic thread scheduling.

3.1.1 SPEEDEX: Towards an Ideal DEX

This paper disproves the conventional wisdom about on-chain DEX performance. We present SPEEDEX, a fully on-chain decentralized exchange that meets all of the desiderata outlined above. SPEEDEX gives every user an equal level of access (thereby eliminating a widespread class of risk-free front-running), eliminates internal arbitrage opportunities (thereby making optimal use of liquidity available on the DEX), and is capable of processing over 200,000 transactions per second when deployed on 48-core machines (Figure 3.3). SPEEDEX is designed to scale further when given more hardware.

Like most blockchains, SPEEDEX processes transactions in blocks—in our case, a block of 500,000 transactions every few seconds. Its fundamental principle is that transactions in a block commute: a block's result is identical regardless of transaction ordering, which enables efficient parallelization [125].

SPEEDEX's core innovation is to execute every order at the same exchange rate as every other order in the same block. SPEEDEX processes a block of limit orders as one unified batch, in which, for example, every 1 EUR sold to buy USD receives exactly 1.10 USD in payment. Furthermore, SPEEDEX's exchange rates present no arbitrage opportunities within the exchange; that is, the exchange rate for trading USD to EUR directly is exactly the exchange rate for USD to YEN multiplied by the rate for YEN to EUR. These exchange rates are unique for any (nonempty) batch of trades. Users interact with SPEEDEX via traditional limit orders, and SPEEDEX executes a limit order if and only if the batch's exchange rate exceeds the order's limit price.

This design provides two additional economic advantages. First, the exchange offers liquid trading between every asset pair. Users can directly trade any asset for any other asset, and the market between these assets will be at least as liquid as the most liquid market path through intermediate reserve currencies. Second, SPEEDEX eliminates a class of front-running that is widespread in modern DEXes. No exchange operator or user with a low-latency network connection can buy an asset and resell it at a higher price, within the same block. (Note that this is not every type of front-running; §3.8 and §3.10 contrast SPEEDEX's guarantees with those of other mitigations, and how they can be combined.)

Furthermore, this economic design enables a scalable systems design that is not possible using traditional order-matching semantics. Unlike every other DEX, the operation of SPEEDEX is efficiently parallelized, allowing SPEEDEX to scale to transaction rates far beyond those seen today. Transactions within a block commute with each other precisely because trades all happen at the same shared set of exchange rates. This means that the transaction processing engine has no need for the sequential read-modify-update loop of traditional orderbook matching engines. Account balances are adjusted using only hardware-level atomics, rather than locking.

3.1.2 SPEEDEX Overview

SPEEDEX is not a blockchain itself; rather, it is a DEX component that can be integrated into any blockchain. A copy of the SPEEDEX module should run inside every replica of a blockchain using the system. SPEEDEX does not depend on any specific property of a consensus protocol, but automatically benefits from throughput advances in consensus and transaction dissemination (such as [144]). SPEEDEX heavily uses concurrency and benefits from uninterrupted access to CPU caches, and as such is best implemented directly within blockchain node software (instead of as a smart contract).

We implemented SPEEDEX within a custom blockchain using the HotStuff consensus protocol [316]; this implementation provides the measurements in this paper. We created a second implementation as a component of the Stellar blockchain[233], which is considering a Layer-1 SPEEDEX deployment.

Implementing SPEEDEX introduces both theoretical algorithmic challenges and systems design challenges. The core algorithmic challenge is the computation of the batch prices. This problem maps to a well-studied problem in the theoretical literature (equilibrium computation of *Arrow-Debreu Exchange Markets*, §A.1.1); however, the algorithms in the theoretical literature scale extremely poorly, both asymptotically and empirically, as the number of open limit orders increases.

We show that the market instances which arise in SPEEDEX have additional structure not discussed in the theoretical literature, and use this structure to build a novel algorithm (based on the Tâtonnement process of [127]) that, in practice, efficiently approximates batch clearing prices. We then explicitly correct approximation error with a follow-up linear program.

Our algorithm's runtime is largely independent of the number of limit orders—each Tâtonnement query has a runtime of $O(\#\text{assets}^2 \cdot \lg(\#\text{offers}))$ and the linear program has size $O(\#\text{assets}^2)$. This gives a crucial algorithmic speedup because in the real world, the number of currencies is much smaller than the number of market participants. (The experiments of §3.6 and §3.7 use 50 assets and tens of millions of open offers.)

On the systems design side, to implement this exchange, we design natural commutative transaction semantics and implement data structures designed for concurrent, batched manipulation and for efficiently answering queries about the exchange state from the price computation algorithm.

In recent years, the economics literature has begun discussing the use of batched trading systems in traditional markets to combat front-running and externalities associated with high-frequency trading [74, 103, 101]. This literature focuses only on the case of trading between two assets (where price computation is simple) or where all trades use a single *numeraire* currency [104]. Our contribution to this line of work is to demonstrate the feasibility of a batch trading system that exchanges many assets and many numeraire currencies simultaneously, thereby expanding the design space of implementable market structures.

3.2 System Architecture

SPEEDEX is an asset exchange implemented as a replicated state machine in a blockchain architecture (Fig. 3.1). Assets are issued and traded by *accounts*. Accounts have public signature keys authorized to spend their assets. Signed transactions are multicast on an overlay network (Fig. 3.1, 1) among block *producers*. At each round, one or more producers propose candidate blocks extending the blockchain history (Fig. 3.1, 2). A set of *validator* nodes (generally the same set or a superset of the producers) validates and selects one of the blocks through a consensus mechanism (Fig. 3.1, 3). SPEEDEX is suitable for integration into a variety of blockchains, but benefits from a consensus layer with relatively low latency (on the order of seconds), such as BA$\star$ [181], SCP [233], or HotStuff [316].

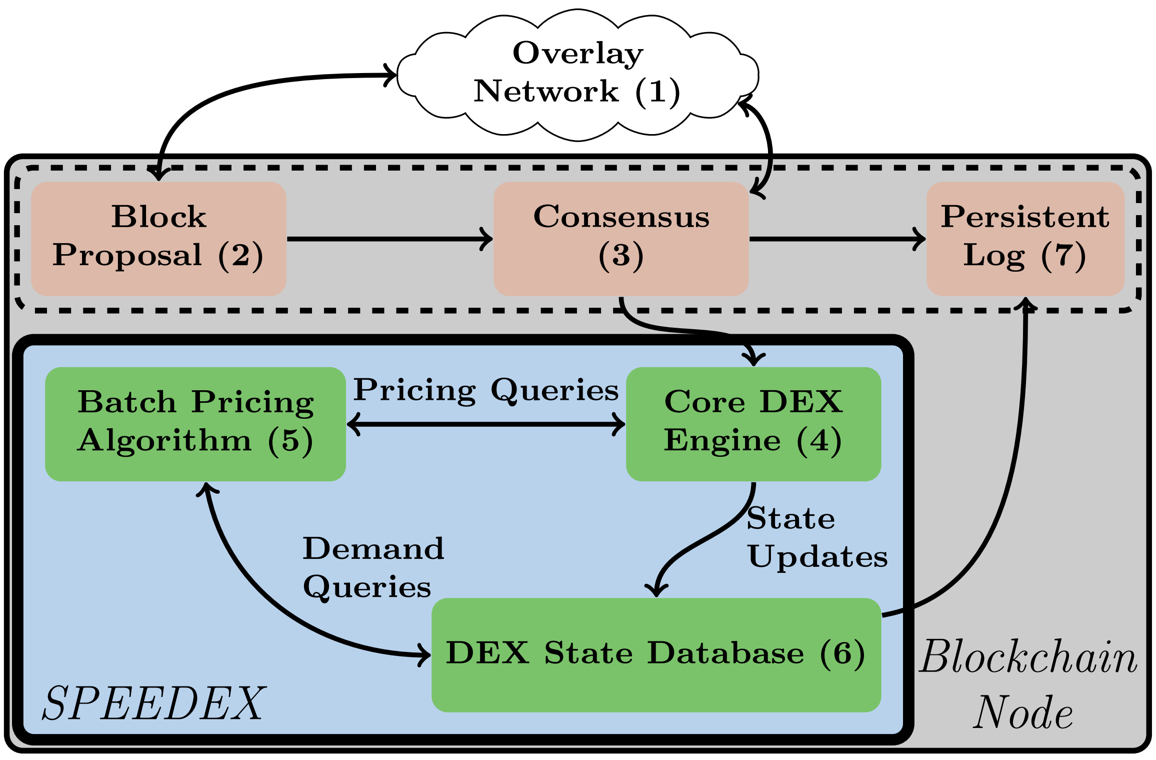


Figure 3.1: Architecture of SPEEDEX module (4, 5, 6) inside one blockchain node.

The implementation evaluated here uses HotStuff [316], while the the Stellar blockchain implementation relies on Stellar's existing consensus protocol, SCP [242].

Most central banks and digital currency issuers maintain a ledger tracking their currency holdings. SPEEDEX is not intended to replace these primary ledgers. Rather, we expect banks and other regulated financial institutions to issue 1:1 backed token deposits onto a blockchain that runs SPEEDEX and provide interfaces for moving money on and off the exchange. These assets could be digital-native tokens as well; any divisible and fungible asset can integrate with SPEEDEX.

SPEEDEX supports four operations: account creation, offer creation, offer cancellation, and send payment. Offers on SPEEDEX are traditional limit orders. For example, one offer might offer to sell 100 EUR to buy USD, at a price no lower than 0.91 USD/EUR. Offers can trade between any pair of assets, in either direction. Another offer, for example, might offer to sell 100 USD in exchange for EUR, at a price no lower than 1.10 EUR/USD.

What makes SPEEDEX different from existing DEXes is the manner in which it processes new orders. Traditional exchanges process trades sequentially, implicitly computing a matching between limit orders. SPEEDEX, by contrast, processes trades in batches (typically, one batch would consist of all of the limit orders in one block of the blockchain).

In a blockchain, all of the transactions in a block are appended at the same clock time, so there is no reason *a priori* why a DEX should pick one ordering over another. SPEEDEX, by design, imposes no ordering whatsoever between transactions in a block. Side effects of a transaction are only visible to other transactions in future blocks.

Logically, when the SPEEDEX core engine (Fig. 3.1, 4) receives a finalized block of trades, it applies all of the trades at exactly the same time and computes an unordered set of state changes, which it passes to its exchange state database (Fig. 3.1, 6). This database records orderbooks and account balances, and is periodically written to the persistent log (Fig 3.1, 7).

3.2.1 SPEEDEX Module Architecture

To implement an exchange that operates replicably where trades in a block are not ordered relative to each other, SPEEDEX requires a set of trading semantics such that operations *commute*.

Traditional exchange semantics are far from commutative: one offer to buy an asset is matched with the lowest priced seller, and the next offer to buy is matched against the second-lowest priced seller, and so on. Hence, every trade can occur at a slightly different exchange rate.

Instead, to make trades commutative, SPEEDEX computes in every block a *valuation* $p_{\mathcal{A}}$ for every asset $\mathcal{A}$. The units of $p_{\mathcal{A}}$ are meaningless, and can be thought of as a fictional valuation asset that exists only for the duration of a single block. However, valuations imply exchange rates between different assets—every sale of asset $\mathcal{A}$ for asset $\mathcal{B}$ occurs at a price of $p_{\mathcal{A}}/p_{\mathcal{B}}$. Unlike traditional exchanges, SPEEDEX does not explicitly compute a matching between trade offers. Instead, offers trade with a conceptual *"auctioneer"* entity at these exchange rates. Trading becomes commutative because all trades in one asset pair occur at the same price.

The main algorithmic challenge is to compute valuations where the exchange *clears*—i.e., the amount of each asset sold to the auctioneer equals the amount bought from the auctioneer.

When the auctioneer sets exact clearing valuations, an offer trades fully with the auctioneer if its limit price is strictly below the auctioneer's exchange rate, and not at all if its limit price exceeds the auctioneers rate. When the limit price equals the exchange rate, SPEEDEX may execute the offer partially. Note that an exchange is a zero-sum system; as compared to sequential execution, some users may see better prices and some may see worse, but SPEEDEX guarantees that no user's price is worse than their minimum limit price.

Theorem 3.2.1. *Exact clearing valuations always exist. These valuations are unique up to rescaling.*[1]

Theorem 3.2.1 is a restatement of a general theorem of Arrow-Debreu exchange market theory [146] (§A.1.3).

Concretely, whenever the core SPEEDEX engine (Fig 3.1, 4) receives a newly finalized block, one of its first actions is to query an algorithm that computes clearing valuations (Fig 3.1, 5). It then uses the output of this algorithm to compute the modifications to the exchange state (Fig 3.1, 6).

As valuations that clear the market always exist for any set of limit orders, there is no adversarial input that SPEEDEX cannot process. And because these valuations are unique, SPEEDEX operators do not have a strategic choice between different sets of valuations. SPEEDEX's algorithmic task is to surface information about a fundamental mathematical property of a batch.

Unfortunately, we are not aware of a practical method to compute clearing prices exactly. (The number of bits required to represent exact clearing prices may be extremely large [146], and in a natural extension of the SPEEDEX model [270] the clearing prices are not even rational.) SPEEDEX therefore uses *approximate* clearing prices.

At nonexact clearing prices, the conceptual auctioneer will not have enough of some asset(s) to pay out all offers willing to accept the market price. SPEEDEX addresses this deficit in two ways. First, the auctioneer proportionally reduces the amount it pays out to offers by a small fraction—in other words, it charges a commission. Commissions are common for exchanges, whether decentralized or not, though SPEEDEX does it for market clearing rather than profit reasons. To avoid incentivizing high trading costs, the implementation returns commissions to the asset issuers, and one goal of our price computation algorithm's design is to make this commission as low as computationally practical. Second, the auctioneer can refrain from filling some marketable offers. Whereas in a perfect Arrow-Debreu exchange market, offers at the market price may be partially filled or not filled, in SPEEDEX the same applies to offers very close to the market price, even if they still beat the market price by a small percentage.

[1] And technical conditions (§A.1.3), e.g. everything clears an empty market.

SPEEDEX always rounds trades in favor of the auctioneer. Our implementation burns collected transaction fees and accumulated rounding error (effectively returning them to the issuer by reducing the issuer's liabilities). The Stellar implementation eliminates the fee and returns the accumulated rounding error to asset issuers.

3.2.2 Design Properties

Computational Scalability SPEEDEX's commutative semantics allow effective parallelization of DEX operation. Because transactions within a block are not semantically ordered, DEX state is identical regardless of the order in which transactions are applied. This exact replicability is, of course, required for a *replicated* state machine. The order-independence also means SPEEDEX transactions can be executed in parallel by all available CPU cores despite the fact that thread interleaving is nondeterministic in multicore machines. Almost all coordination occurs via hardware-level atomics (e.g., atomic add on 64-bit integers) without spinlocks.

SPEEDEX stores balances in accounts, rather than in discrete, unspent coins (often called "UTXOs"). It also supports single-currency payment operations, which are simpler than DEX trading. Hence, SPEEDEX disproves the popular belief [234, 272] that account-based ledgers are not compatible with horizontal scalability.

No risk-free front running Well-placed agents in real-world financial markets can spy on submitted offers, notice a new transaction T, and then submit a transaction T' (that executes before T) that buys an asset and re-sells it to T at a slightly higher price. In some blockchain settings, T' can be done as a single atomic action [143]. However, since every transaction sees the same clearing prices in SPEEDEX, back-to-back buy and sell offers would simply cancel each other out. Relatedly, because every offer sees the same prices, a user who wishes to trade immediately can set a very low minimum price and be all but guaranteed to have their trade executed, but still at the current market price.

Risk-free front-running is one instance of the widely discussed "Miner Extractable Value" (MEV) [143] phenomenon, in which block producers reorder transactions within a block for their own profit (or in exchange for kickbacks). By eliminating the ordering of transactions within a block, SPEEDEX eliminates a large source of MEV. However, this does not eliminate every type of front-running manipulation, such as delaying victim transactions to a future block (see §3.8).

No (internal) arbitrage and no central reserve currency An agent selling asset $\mathcal{A}$ in exchange for asset $\mathcal{B}$ will see a price of $p_{\mathcal{A}}/p_{\mathcal{B}}$. An agent trading $\mathcal{A}$ for $\mathcal{B}$ via some intermediary asset $\mathcal{C}$ will see exactly the same price, as $\frac{p_{\mathcal{A}}}{p_{\mathcal{C}}} \cdot \frac{p_{\mathcal{C}}}{p_{\mathcal{B}}} = \frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$. Hence, one can efficiently trade between assets without much pairwise liquidity with no need to search for an optimal path. By contrast, many international payments today go through USD because of a lack of pairwise liquidity. The multitude of USD-pegged stablecoins in modern blockchains further fragments liquidity. Of course, there can still be arbitrage between SPEEDEX and external markets.

3.3 Commutative DEX Semantics

To propose or execute a block of transactions, the SPEEDEX core engine performs the following three actions.

1 For each transaction in the block (in parallel), check signature validity, collect new limit offers, and compute available account balances after funds are committed to offers or transferred between accounts. When proposing a block of transactions, SPEEDEX discards potentially invalid transations.

2 When proposing a block, compute approximate clearing prices and approximation correction metadata.

3 Iterate over each offer, making a trade or adding it to the resting orderbooks (based on the prices and metadata).

For transaction processing in step 1 to be commutative, it must be the case that the step 1 output effects (specifically: create a new account, create a new offer, cancel an existing offer, and send a payment) of one transaction have no influence on the output effects of another transaction. This means that one transaction cannot read some value that was output by another transaction (in the same block), and that whether one transaction succeeds cannot depend on the success of another transaction.

To meet the first requirement, traders include all parameters to their transactions within the transaction itself. The second requirement necessitates precise management of transaction side effects. At most one transaction per block may alter an account's metadata (such as the account's public key or existence), and metadata changes take effect only at the end of block execution. Similarly, an offer cannot be created and cancelled in the same block. As payments and trading are the common case, we do not consider these restrictions a serious limitation.

SPEEDEX must also ensure that no account is overdrafted. That is to say, after processing all transactions in a block, the unlocked balance of every account must be nonnegative (where an open offer locks the offered amount of an asset for the duration of its lifetime). Unlike most distributed ledgers, SPEEDEX cannot simply deem the second of two conflicting transactions to fail—after all, transactions have no ordering. Instead, our implementation requires a block proposer to ensure that a block cannot cause overdrafts; every node rejects blocks that violate this property. To generate valid blocks, proposers use a conservative process outlined in §A.11.6. The design requires passing information from the SPEEDEX database (Fig 3.1, 6) to the proposal module (Fig 3.1, 2).

The core remaining technical challenge is the batch price computation (Fig 3.1, 5).

3.4 Price Computation

3.4.1 Requirements

As discussed earlier, in every block, SPEEDEX computes batch clearing prices and executes trades in response to these prices. Every DEX is subject to two fundamental constraints:

- **Asset Conservation** No assets should be created out of nothing. As discussed in §3.2, offers in SPEEDEX trade with a virtual auctioneer. After a batch of trades, this auctioneer cannot be left with any debt. We do allow the auctioneer to burn some surplus assets as a fee.
- **Respect Offer Parameters** No offer trades at a worse price than its limit price.

Additionally, SPEEDEX should facilitate as much trade volume as possible. (Otherwise, the constraints could be vacuously met by never trading.) Furthermore, price computation must be efficient, as it occurs for each block of trades, every few seconds. Finally, SPEEDEX should minimize the number of offers that trade partially; asset quantities are stored as integer multiples of a minimum unit, so each partial trade risks accumulating a rounding error of up to one unit.

3.4.2 From Theory To Practice

The problem of computing batch clearing prices is equivalent to the problem of computing equilibria in linear Arrow-Debreu Exchange Markets (§A.1). Our algorithm is based on the iterative Tâtonnement process from this literature [127].

However, the runtimes of the theoretical algorithms scale very poorly, both asymptotically and empirically. They also output approximate equilibria for notions of approximation that violate the two fundamental constraints above (for example, Definition 1 of [127] permits equilibria to mint new assets and to steal from a user).

We develop a novel algorithm for computing equilibria that runs efficiently in practice (§3.6) and explicitly ensures that (1) asset amounts are conserved and (2) every offer trades at exactly the market prices, and only if the offer's limit price is at or below the batch exchange rate. First, Tâtonnement approximates clearing prices (§3.5). We show that the structure of the types of trades in SPEEDEX lets each iteration run in time logarithmic in the number of open limit offers (via a series of binary searches), giving an algorithm asymptotically faster than that within the theoretical literature.

We then explicitly correct for the approximation error with a linear program (§A.4). Crucially, the size of this linear program is linear in the number of asset pairs, and has no dependence on the number of open trade offers. The linear program ensures that, no matter what prices Tâtonnement outputs, (1) asset amounts are conserved, and (2) no offer trades if the batch price is less than its limit price.

To be precise, our algorithm outputs the following:

- **Prices:** For each asset $\mathcal{A}$, SPEEDEX computes an asset valuation $p_{\mathcal{A}}$. One unit of $\mathcal{A}$ trades for $p_{\mathcal{A}}/p_{\mathcal{B}}$ units of $\mathcal{B}$.
- **Trade Amounts:** For each asset pair $(\mathcal{A}, \mathcal{B})$, SPEEDEX computes an amount $x_{\mathcal{A},\mathcal{B}}$ of asset $\mathcal{A}$ that is sold for asset $\mathcal{B}$ (again, at exchange rate $p_{\mathcal{A}}/p_{\mathcal{B}}$).

For every asset pair $(\mathcal{A}, \mathcal{B})$, SPEEDEX sorts all of the offers selling $\mathcal{A}$ for $\mathcal{B}$ by their limit prices, and then executes the offers with the lowest limit prices, until it reaches a total amount of $\mathcal{A}$ sold of $x_{\mathcal{A},\mathcal{B}}$ (tiebreaking by account ID and offer ID).

As a bonus, this method ensures that at most one offer per trading pair executes partially, minimizing rounding error.

3.5 Price Computation: Tâtonnement

Tâtonnement is an iterative process; starting from an (arbitrary) initial set of prices, it iteratively refines them until the prices reach a stopping criterion.

Each iteration of Tâtonnement starts with a *demand query.* The *demand* of an offer is the net trading of the offer (with the auctioneer) in response to a set of prices, and the demand of a set of offers is the sum of the demands of each offer. Tâtonnement's goal is to find prices such that the amount of each asset sold to the auctioneer matches the amount bought from it (in other words, the net demand is 0).

Example 3.5.1. *Suppose that a limit order offers to sell 100 USD for EUR with a minimum price of* 0.8 *EUR per USD. If the candidate prices are such that* $\alpha = \frac{p_{USD}}{p_{EUR}} > 0.8$*, then the limit order would like to trade, and its demand is* $(-100\ USD, 100\alpha\ EUR)$*. Otherwise, its demand is* $(0\ USD, 0\ EUR)$*.*

Iterative Price Adjustment. If more units of an asset are demanded from the auctioneer than are supplied to it (a positive net demand, meaning a deficit for the auctioneer), then the auctioneer raises the price of the asset. Otherwise, the auctioneer has a surplus, so it lowers the price of the asset. Implementing this process effectively requires careful numerical normalization in response to differences in prices and trade volumes, which we describe in detail in §A.3.1.

Tâtonnement repeats this process until the current set of prices is sufficiently close to the market clearing prices (or it hits a timeout). Specifically, Tâtonnement iterates until it has a set of prices such that, if the auctioneer charges a commission of ε, then there is a way to execute offers such that:

1 The auctioneer has no deficits (assets are conserved)

2 No offer executes outside its limit price bound

3 Every offer with a limit price more than a $(1-\mu)$ factor below the auctioneer's exchange rate executes in full.

The last condition is a formalization of the notion that SPEEDEX should satisfy as many trade requests as possible. Informally, an offer with a limit price equal to the auctioneer's exchange rate is indifferent between trading and not trading, while one with a limit price far below the auctioneer's exchange rate strongly prefers trading to not trading.

3.5.1 Efficient Demand Queries

Implemented naïvely, Tâtonnement's demand queries would consist of a loop over every open exchange offer. This is impossibly expensive, even if the loop is massively parallelized. Concretely, one invocation of Tâtonnement can require many thousands of demand queries. Every demand query therefore must return results in at most a few hundred microseconds.

This naïve loop appears to be required for the (more general) problem instances studied in the theoretical literature. However, all of the offers in SPEEDEX are traditional limit orders that sell one asset in exchange for one other asset at some limit price. An offer with a lower limit price always trades if an offer with a higher limit price trades. Therefore, SPEEDEX groups offers by asset pair and sorts offers by their limit prices. We drive the marginal cost of this sorting to near zero by using an offer's limit price as the leading bits (in big-endian) of the keys in our Merkle tries (§A.11.5).

SPEEDEX can therefore compute a demand query with a sequence of binary searches (§A.7). Individual binary searches can run on separate CPU cores. The number of open offers (say, M) on an exchange is vastly higher than the number of assets traded (say, N). Our experiments in §3.7 trade $N = 50$ assets with $M =$ tens of millions of open offers; the complexity reduction from $O(M)$ to $O(N^2 \lg(M))$ is crucial.

3.5.2 Multiple Tâtonnement Instances

§A.3 describes several other Tâtonnement adjustments that help it respond well to a wide variety of market conditions. Some of these adjustments are parametrized (such as how quickly one should adjust the candidate prices); rather than pick one set of control parameters, we run several instances of Tâtonnement in parallel and take whichever finishes first as the result. (In the case of a timeout, we choose the set of prices that minimizes the *unrealized utility* [§3.6.2].) SPEEDEX includes the output of Tâtonnement and the subsequent linear program in the headers of proposed blocks (§A.11.3).

3.6 Evaluation: Price Computation

Tâtonnement's runtime depends primarily on the target approximation accuracy, the number of open trade offers, and the distribution of the open trade offers. The runtime increases as the desired accuracy increases. Surprisingly, the runtime actually *decreases* as the number of open offers increases. And like many optimization problems, Tâtonnement performs best when the input is normalized, meaning in this case that the (normalized, §A.3.1) volume traded of each asset is roughly the same.

Tâtonnement runs once per block. To produce a block every few seconds, Tâtonnement must run in under one second most of the time. Our implementation runs Tâtonnement with a timeout of 2 seconds, but it typically converges much faster.

3.6.1 Accuracy and Orderbook Size

We find that Tâtonnement converges more quickly as the number of open offers *increases.* Tâtonnement converges fastest when small price changes do not cause comparatively large changes in overall net demand. However, an offer's behavior is a discontinuous function (of prices); it does not trade below its limit price and trades fully above it.

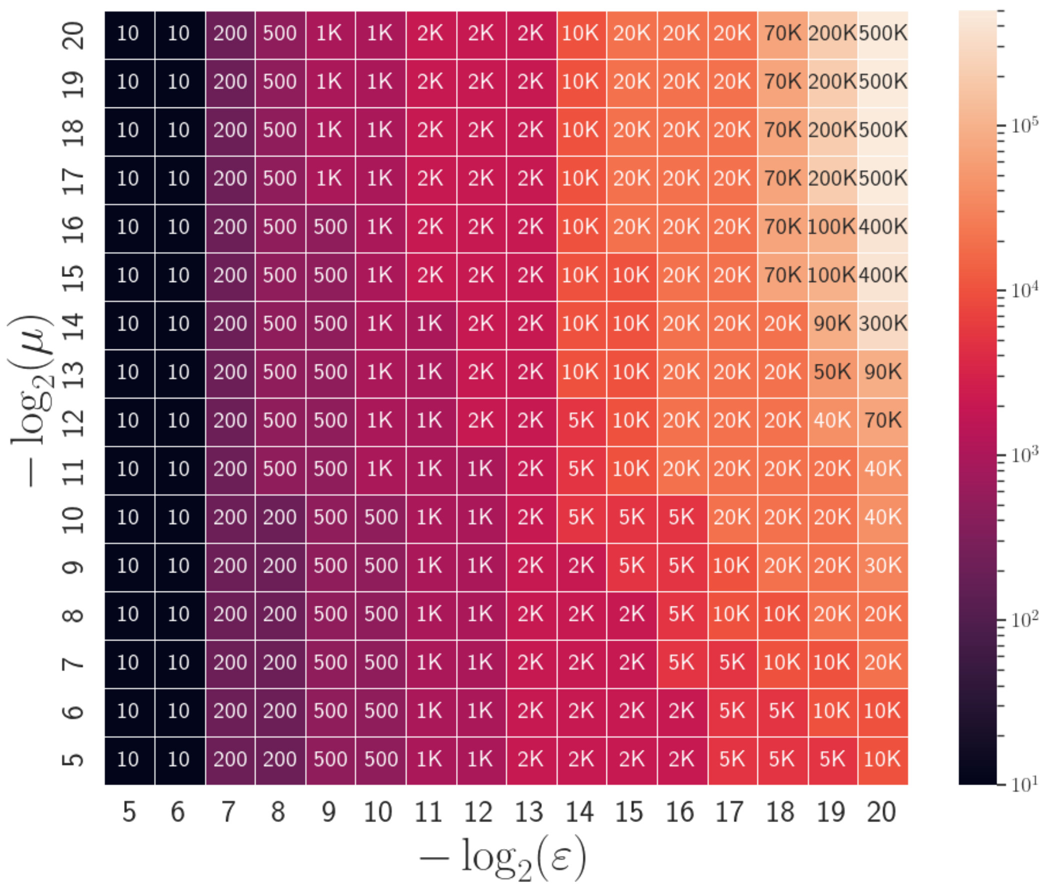
There are two factors that mitigate these "jump discontinuities." First, Tâtonnement approximates optimal offer behavior by a continuous function (§A.2). Smaller μ means a closer approximation. Second, the more offers there are in a batch, the smaller any one offer's relative contribution to overall demand. This last factor explains why Tâtonnement converges more quickly when there are more offers on the exchange. A real-world deployment might raise accuracy as trading increases.

Fig. 3.2 plots the minimum number of trade offers that Tâtonnement needs to consistently find clearing prices for 50 distinct assets in under 0.25 seconds (for the same trade distribution used in §3.7). To put these fee rates in context, BinanceDex [2] charged a fee of either $0.1\% \approx 2^{-10}$ or $0.04\% \approx 2^{-11.3}$. Uniswap [55, 56] charges 1%, 0.3%, or 0.05% ($\sim 2^{-6.6}$, $\sim 2^{-8.4}$, and $\sim 2^{-11}$, respectively), and Coinbase charges 0.5% to 4% [5] ($\sim 2^{-7.6}$ to $\sim 2^{-4.6}$).

Though our experiments rarely experienced Tâtonnement timeouts, Tâtonnement timeouts caused by sparse orderbooks may be self-correcting: If SPEEDEX proposes suboptimal prices, fewer offers will find a counterparty and trade. When fewer offers clear in one block, more are left to facilitate Tâtonnement in the next block. §A.6 describes an alternative algorithm that is effective on small batches.

3.6.2 Robustness Checks

As a robustness check, we run Tâtonnement against a trade distribution derived from volatile cryptocurrency market data. In an ideal world, we could replay trades from another DEX through SPEEDEX. Unfortunately, doing so poses several problems. First, in practice, almost all DEX trades go through four de facto reserve currencies (ETH, USD, USDC, and USDT), three of which are always worth close to $1. The decomposition between a few core "pricing" assets and a larger number of other assets makes price discovery too simple. Second, transaction rates on existing DEXes are too low to provide enough data. Finally, we suspect users would submit different orders



	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
20	10	10	200	500	1K	1K	2K	2K	2K	10K	20K	20K	20K	70K	200K	500K
19	10	10	200	500	1K	1K	2K	2K	2K	10K	20K	20K	20K	70K	200K	500K
18	10	10	200	500	1K	1K	2K	2K	2K	10K	20K	20K	20K	70K	200K	500K
17	10	10	200	500	1K	1K	2K	2K	2K	10K	20K	20K	20K	70K	200K	500K
16	10	10	200	500	500	1K	2K	2K	2K	10K	20K	20K	20K	70K	100K	400K
15	10	10	200	500	500	1K	2K	2K	2K	10K	10K	20K	20K	70K	100K	400K
14	10	10	200	500	500	1K	1K	2K	2K	10K	10K	20K	20K	20K	90K	300K
13	10	10	200	500	500	1K	1K	2K	2K	10K	10K	20K	20K	20K	50K	90K
12	10	10	200	500	500	1K	1K	2K	2K	5K	10K	20K	20K	20K	40K	70K
11	10	10	200	500	500	1K	1K	1K	2K	5K	10K	20K	20K	20K	20K	40K
10	10	10	200	200	500	500	1K	1K	2K	5K	5K	5K	20K	20K	20K	40K
9	10	10	200	200	500	500	1K	1K	2K	2K	5K	5K	10K	20K	20K	30K
8	10	10	200	200	500	500	1K	1K	2K	2K	2K	5K	10K	10K	20K	20K
7	10	10	200	200	500	500	1K	1K	2K	2K	2K	5K	5K	10K	10K	20K
6	10	10	200	200	500	500	1K	1K	2K	2K	2K	2K	5K	5K	10K	10K
5	10	10	200	200	500	500	1K	1K	2K	2K	2K	2K	5K	5K	5K	10K

Figure 3.2: Minimum number of offers needed for Tâtonnement to run in under 0.25 seconds (Smaller is better. Times averaged over 5 runs). The x axis denotes offer behavior approximation quality (μ), and the y axis denotes the commission (ε).

to SPEEDEX than they might on a traditional exchange, due to the distinct economic properties of batch trading systems.

Experiment Setup. As a next-best alternative, we generate a dataset based on historical price and market volume data. We took the 50 crypto assets that had the largest market volume on December 8, 2021 (as reported by `coingecko.com`) and for each asset, gathered 500 days of price and trade volume history. We then generated 500 batches of 50,000 transactions. A new offer in batch i sells asset $\mathcal{A}$ (and buys asset $\mathcal{B}$) with probability proportional to the relative volume of asset $\mathcal{A}$ (and asset $\mathcal{B}$, conditioned on $\mathcal{A} \neq \mathcal{B}$) on day i, and demands a minimum price close to the real-world exchange rate on day i. The extreme volatility of cryptocurrency markets and variation between these 50 assets make this dataset particularly difficult for Tâtonnement. To further challenge Tâtonnement, we use a smaller block size of $\sim 30,000$ (compared to $500,000$ in §3.7).

The experiment charged a commission of $\varepsilon = 2^{-15} \approx 0.003\%$, and attempted to clear offers with limit prices more than $1 - \mu$ below the market prices, for $\mu = 2^{-10} \approx 0.1\%$ (§A.2).

Experiment Results. The experiment ran for 500 blocks. Each block created about 25,000 new offers and a few thousand cancellations and payments.

Tâtonnement computed an equilibrium quickly in 350 blocks, and in the remainder, computed prices sufficiently close to equilibrium that the follow-up linear program facilitated the vast majority of possible trading activity.

We measure the quality of an approximate set of prices by the ratio of the "unrealized utility" to the "realized utility." The utility gained by a trader from selling one unit of an asset is the difference between the market exchange rate and the trader's limit price, weighted by the valuation of the asset being sold. Note that the units do not matter when comparing relative amounts of "utility."

In the blocks where Tâtonnement computed an equilibrium quickly, the mean ratio of unrealized to realized utility was 0.71% (max: 4.7%), and in the other blocks, the mean ratio was 0.42% (max: 3.8%).

Recall that Tâtonnement terminates as soon as a stopping criteria is met; roughly, "does the supply of every asset exceed demand," so one mispriced asset will cause Tâtonnement to keep running. However, every Tâtonnement iteration continues to refine the price of every asset. This is why Tâtonnement actually gives more accurate results in the batches it found challenging. A deployment might enforce a minimum number of Tâtonnement rounds.

Qualitatively, Tâtonnement correctly prices assets with high trading volume and struggles on sparsely traded assets (as might be expected from Fig. 3.2). Tâtonnement also adjusts its price adjustment rule in response to recent market conditions (§A.3.1), a tactic which is less effective on volatile assets.

Should this pose a problem in practice, a deployment could choose to vary the approximation parameters by trading pair.

3.7 Evaluation: Scalability

We ran SPEEDEX on four r6id.24xlarge instances in an Amazon Web Services datacenter. Each instance has 48 physical CPU cores divided over two Intel Xeon Platinum 8375CL chips (32 total cores per socket, 24 of which are allocated to our instances), running at 2.90Ghz with hyperthreading enabled, 768GB of memory, 4 1425GB NVMe drives connected in a RAID0 configuration. We use the XFS filesystem [290]. These experiments use the HotStuff consensus protocol [316], and do not include Byzantine replicas or a rotating leader.

Experiment Setup. These experiments simulate trading of 50 assets. Transactions are charged a fee of $\varepsilon = 2^{-15}(0.003\%)$. We set $\mu = 2^{-10}$, guaranteeing full execution of all orders priced below 0.999 times the auctioneer's price. The initial database contains 10 million accounts. Tâtonnement never timed out, and typically required fewer than 1,000 iterations.

Transactions are generated according to a synthetic data model—every set of 100,000 transactions is generated as though the assets have some underlying valuations, and users trade a random asset pair using a minimum price close to the underlying valuation ratio. The valuations are modified (via a geometric Brownian motion) after every set. Accounts are drawn from a power-law distribution.

Each set is split into four pieces, with one piece given to each replica. Replicas load these sets sequentially and broadcast each set to every other replica. Each replica adds received transactions to its pool of unconfirmed transactions.

Replicas propose blocks of roughly 500,000 transactions. In these experiments, each block consists of roughly 350,000–400,000 new offers, 100,000–150,000 cancellations, 10,000–20,000 payments, and a small number of new accounts. We generate 5,000 sets of input transactions. Some of these transactions conflict with each other and are discarded by SPEEDEX replicas. Each experiment runs for 700–750 blocks.

Every five blocks, the exchange commits its state to persistent storage in the background (via LMDB [124], §A.11.2).

Performance Measurements. Fig. 3.3 plots the end-to-end transaction throughput rate of SPEEDEX as the number of worker threads inside SPEEDEX increases. The x-axis plots the number of open offers on the exchange.

Most importantly, Fig. 3.3 demonstrates that SPEEDEX can efficiently use its available CPU hardware. The speedup is near-linear, until the number of threads approaches the number of CPU cores—from 6 to 12, $\sim 1.9x$, from 12 to 24, $\sim 1.8x$, and from 24 to 48, $\sim 1.4x$. The thread counts are only for the number of threads directly for SPEEDEX's critical path, and not for many of the tasks that the implementation must perform in the background, such as logging data to persistent storage (logging the account database uses 16 threads), consensus, and garbage collection, and these threads begin to contend with SPEEDEX as the number of SPEEDEX worker threads increases.

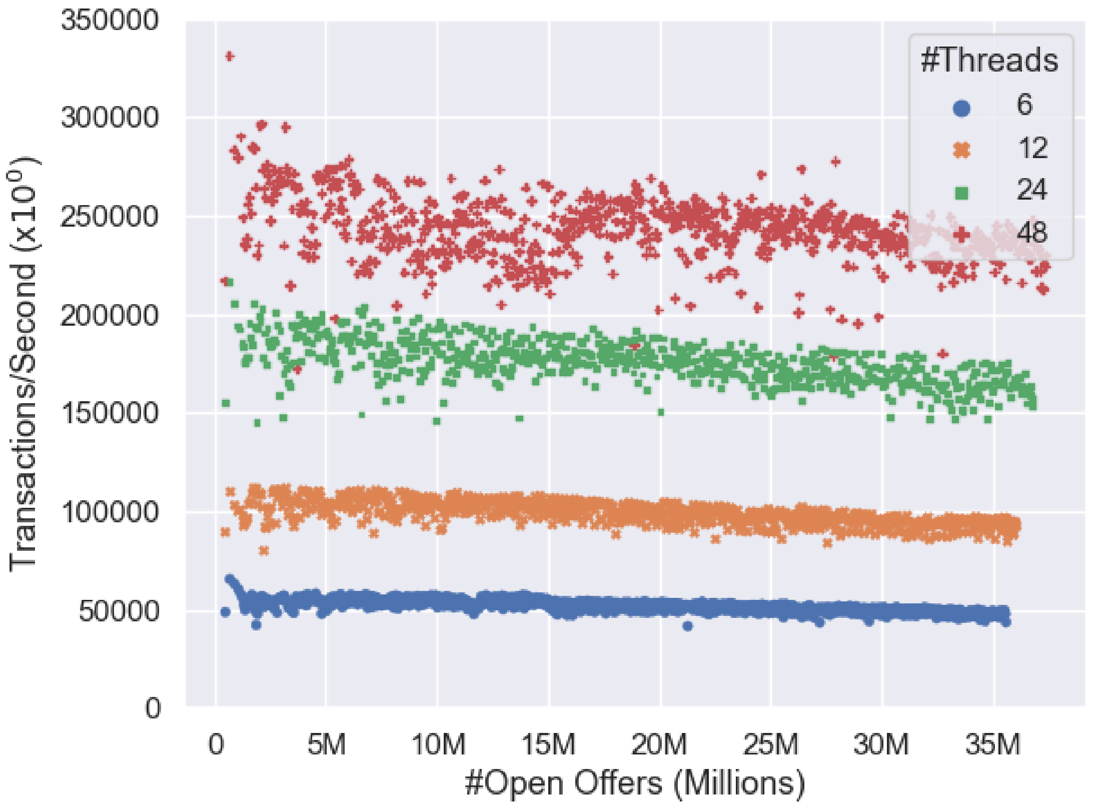


Figure 3.3: Transactions per second on SPEEDEX, plotted over the number of open offers.

Secondly, Fig. 3.3 demonstrates the scalability of SPEEDEX with respect to the number of open offers. The number of open offers SPEEDEX works with in these experiments is already quite large, but most importantly, as the number of open offers goes from 0 to the 10s of millions, SPEEDEX's transaction throughput falls by only $\sim 10\%$. This slowdown is primarily derived from a Tâtonnement optimization (the precomputation outlined in §3.9.2). Tâtonnement is the one part of SPEEDEX that cannot be arbitrarily parallelized, so we design our implementation towards making it as fast as possible. An implementation might skip this work in some parameter regimes.

To focus on the performance of SPEEDEX, Figs. 3.4 and 3.5 plot the time to propose and execute blocks, and to validate and execute proposals, respectively, when we disable signature verification (which is trivial to parallelize). First, note that both proposal and validation scale with the number of threads; validation scales better than proposal due to the aforementioned Tâtonnement optimization. Second, note that validating and executing a proposal from another replica is substantially faster than proposing a block; this lets a replica that is somehow delayed catch up.

The runtime variation in Fig. 3.4 results from the fact that SPEEDEX without signature verification runs too quickly for our persistent logging implementation.

SPEEDEX is not a consensus protocol, and these experiments (one consensus invocation every few seconds) do not come close to stressing the consensus throughput of Hotstuff. However, network bandwidth requirements necessarily scale (at least) linearly with transaction rate. Recent work, such as [144, 206, 314], develops consensus protocols that maximally use available network bandwidth. However, integrating SPEEDEX with any consensus protocol requires understanding the tradeoffs between batch size, transaction rate, and consensus frequency. Fig. 3.6 plots this tradeoff running SPEEDEX on the same transaction workload as in Fig. 3.3. We also ran SPEEDEX with more replicas on different hardware and observed the same scalability trends, as outlined in §A.12 (albeit with lower overall throughput on weaker hardware).

Conclusions. To reiterate, SPEEDEX achieves these transaction rates while operating fully on-chain, with no offchain rollups and no sharding of the exchange's state. To make SPEEDEX faster, one can simply give it more CPU cores, without changing the transaction semantics or user interface. This scaling property is unique among existing DEXes.

3.7.1 Alternative Scaling Techniques

Traditional Exchange Semantics. The core logic of just an exchange system can be implemented extremely efficiently with almost no code. The logic of the constant product market maker UniswapV2 [55], for example, is less than 10 lines of simple arithmetic code. An orderbook-based exchange requires more code but can still be made very fast, as most operations modify only a small number of data objects. We implemented a bare-bones orderbook exchange with two assets using

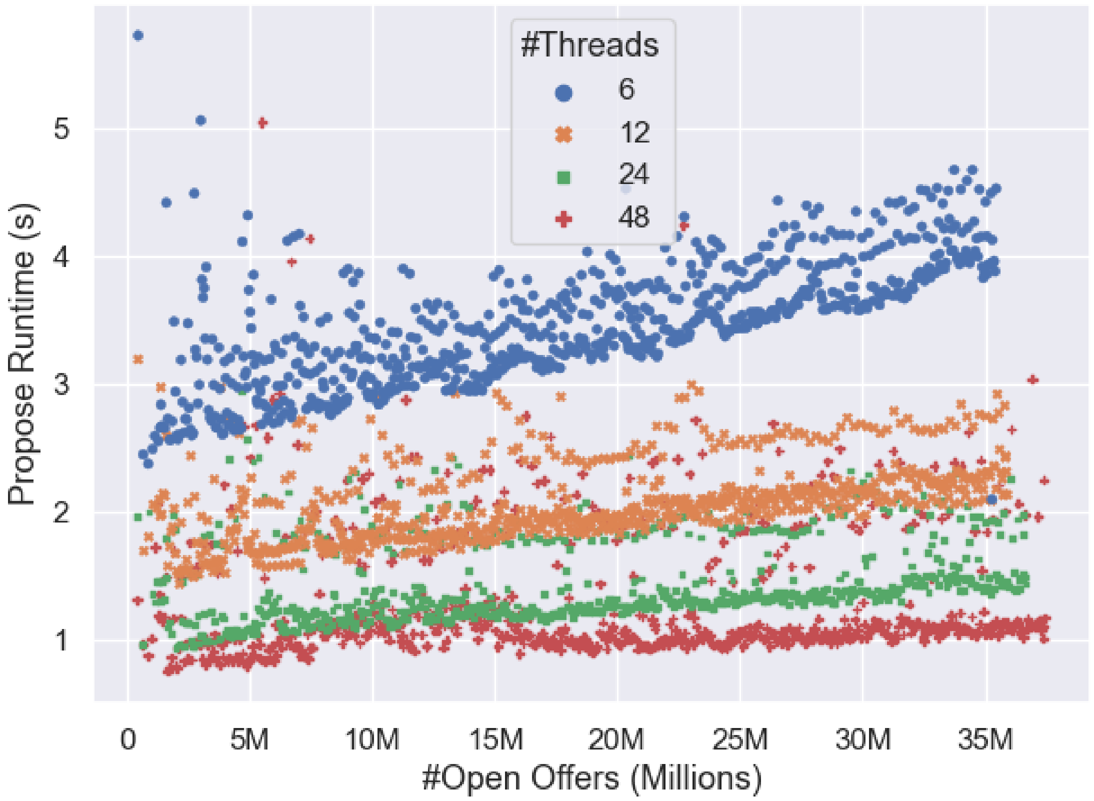


Figure 3.4: Time to propose and execute a block, plotted over the number of open offers.

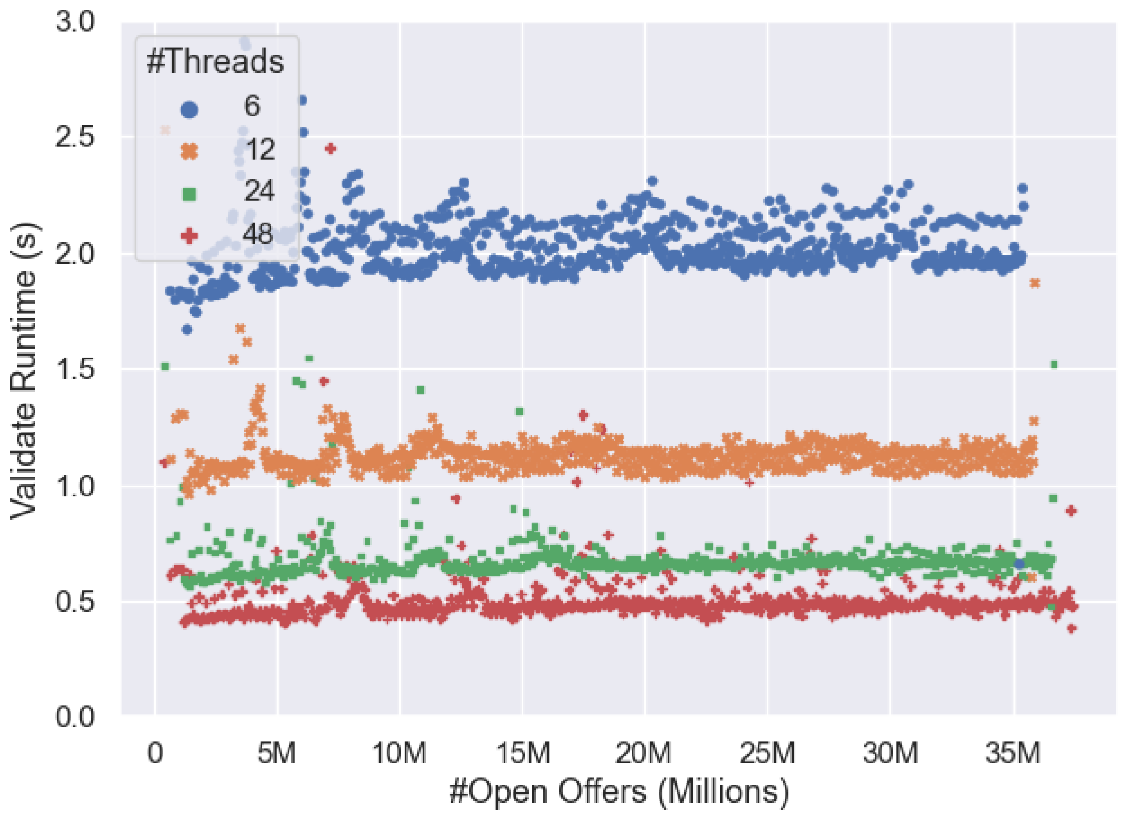


Figure 3.5: Time to validate and execute a proposal, plotted over the number of open offers (measurements from one replica).

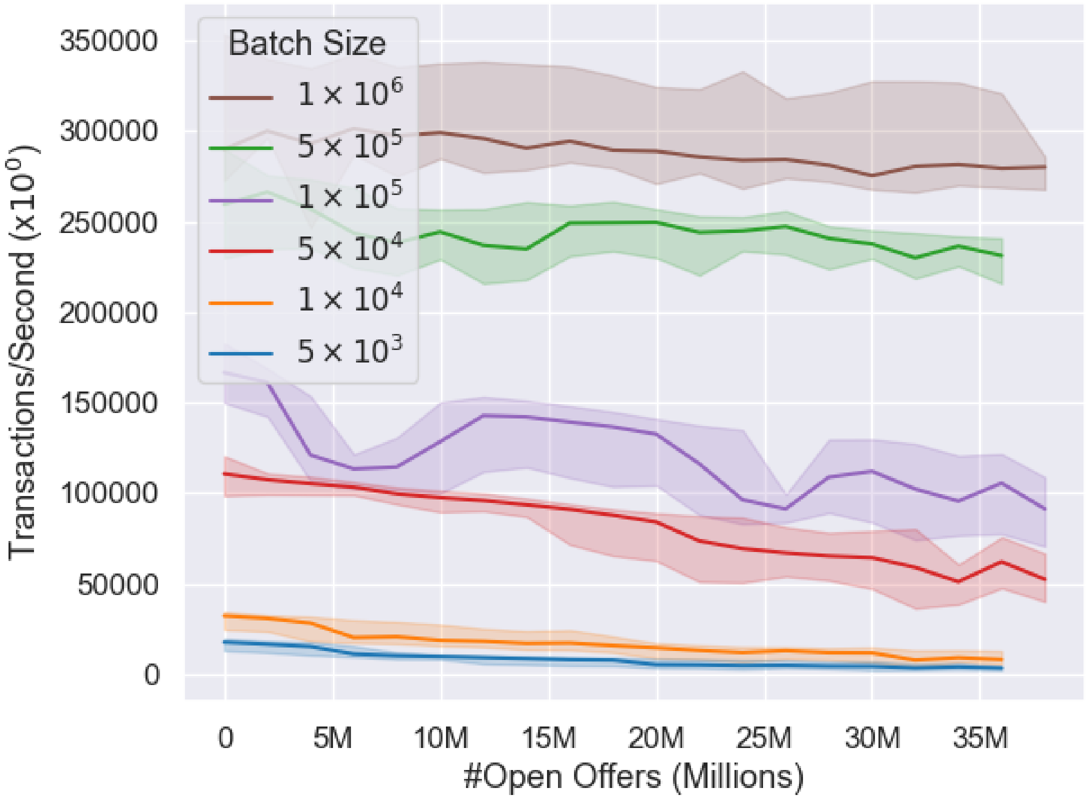


Figure 3.6: Median transaction rates, varying block size and number of open offers (grouped into buckets of 2M). Shaded areas plot 10th to 90th percentiles.

the same data structures as in SPEEDEX—each transaction checks the orderbook for a matching offer or offers and either makes appropriate transfers or adds the new offer to the orderbook. These operations are extremely fast when the number of accounts is small; our implementation runs ~ 1.7 million of these transactions per second when there are only 100 accounts. However, every database lookup becomes slower as the as the number of accounts grows; when there are 10 million accounts in the database (as in the above experiments), throughput falls 8x to $\sim 210,000$ per second. Yet that is before adding all of the other SPEEDEX features one needs in a real DEX, such as state hashes, transaction fees, structures for simple payment verification [249], replication, or durable logging. The scalability of the full SPEEDEX implementation lets it surpass that rate even when slowed down by all of these features.

Note that every orderbook operation affects every subsequent transaction—each transaction influences the exchange rate observed in the next transaction—and as such, their execution cannot be parallelized. SPEEDEX's design, therefore, enables parallel execution of what would otherwise be a strictly serial workload. To isolate the effect of SPEEDEX's parallelizable semantics on its transaction throughput, we therefore turn to a workload that does not touch the DEX at all—one where every transaction is a payment between random accounts.

Optimistic Concurrency Control. A widely explored class of alternative designs for parallel transaction execution use optimistic concurrency control, and of these approaches the most closely related state of the art design appears to be Block-STM [179], which is deployed in Aptos [36]. This approach optimistically executes batches of transactions, retrying after conflicts as necessary.

We therefore design the measurements of Fig. 3.7 to mirror the experiments in [179]. The "Aptos p2p" transactions in [179] are payments between two random accounts, and consist of 8 reads of 5 writes. Each of our payments consists of two data reads (source account public key and last committed sequence number), two atomic compare_exchange operations (subtract payment and fee from source), an atomic fetch_xor (reserve sequence number), and an atomic fetch_add (add payment to destination)—implemented without atomics, this would be 6 reads and 4 writes. All payments are of the same asset.

Fig. 3.7 plots the throughput rates of SPEEDEX on this transaction workload for the parameter settings measured in Block-STM (Figs. 7 and 8, [179]). Note that for large batch sizes, the transaction throughput is largely independent of the number of accounts, even though every transaction in the two account setting contends with every other transaction. Furthermore, unlike Block-STM, SPEEDEX achieves near-linear scalability on sufficiently large batches. For small batch sizes, a large number of accounts actually slows down SPEEDEX, largely due to increased sensitivity to cache performance and our system's NUMA (two socket) architecture on small timescales. We also ran this experiment on a single-socket system (an AWS c5a.16xlarge, as in [179]), and found only negligle impact of the number of accounts on throughput. Fig. 3.7 was run with hyperthreading disabled, to compare against Block-STM experiments. The rest of our experiments were run with

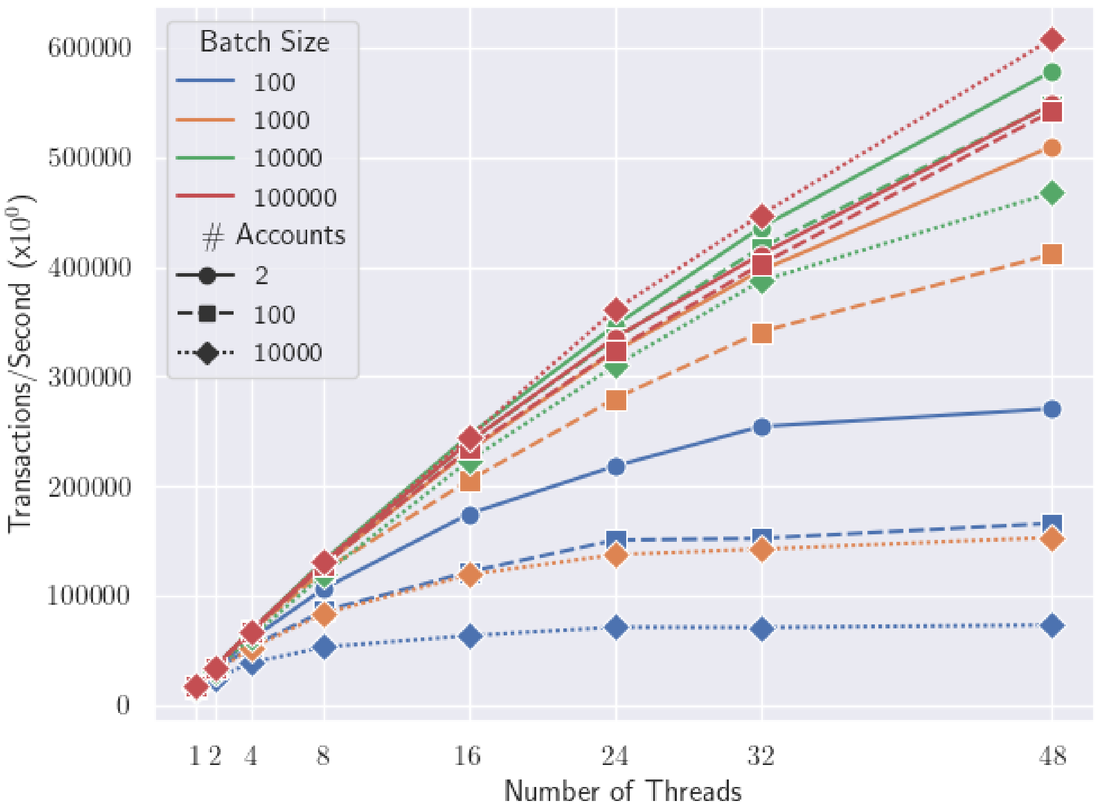


Figure 3.7: Throughput of SPEEDEX on batches of payment transactions with varying thread counts (average of 100 trials).

hyperthreading enabled (because of the many background tasks in SPEEDEX); enabling hyperthreading on this payments workload causes a negligible performance degradation for large batches (approximately 1-6%), and a larger (up to 25%) on small batches. As a baseline comparison, §A.10 graphs the performance of Block-STM on these parameter settings on our hardware.

We also ran SPEEDEX on an only-payments workload with 10 million accounts and 50 assets, and measured a throughput of approximately 375k, 215k, 114k, and 60k transactions per second using 48, 24, 12, and 6 threads, respectively (a 34.8x, 20.0x, and 10.6x, and 5.6x speedup over the single-threaded measurement). We disabled data persistence for these trials—again, the logging off of the critical path contends with SPEEDEX at these transaction rates, especially for payment transactions that modify two accounts, instead of just one (as when creating an offer). The throughput reached 255k transactions per second with data persistence enabled.

Production Systems. Finally, we ran the Ethereum Virtual Machine (Geth 1.10 [16]) on a workload of UniswapV2 [55] transactions, and measured a rate of $\sim$ 3000 transactions per second (a result in line with other Ethereum benchmarks [303]). The Loopring exchange, built as an L2 rollup on Ethereum, claims a maximum rate of $\sim$ 2000 per second [39], a number calculated from Ethereum's per-block computation limit [35], which is in turn set based on the real computational cost of serial transaction execution [261, 108, 117, 59]. Precise measurements of the Stellar blockchain's orderbook DEX suggest that its implementation could handle $\sim$ 4000 DEX trades per second.

3.8 Design Limitations and Mitigations

Latency. Batch trading inherently introduces latency (between order submission and order execution) not present on traditional, centralized exchanges, simply because an order cannot execute until a batch has been closed and clearing prices have been computed. This latency is already present in a blockchain context (a transaction is not finalized until the consensus protocol adds it to a block), so in this context, SPEEDEX introduces no additional latency.

The latency may have downstream economic effects. Market-making may be more (or less) profitable operating in a batch system, which could lead to reduced (or increased) liquidity. Budish et al. [74] argue that batch trading (between 2 assets) would reduce costs for market-makers, which could lead to increased liquidity. However, they study a higher batch frequency (approximately once per millisecond); our lower batch frequency is less studied (see Q9, [101]).

Tâtonnement Nondeterminism. The algorithms evaluated in §3.6 can be viewed as a randomized approximation scheme, which raises the question of whether a malicious operator can manipulate the approximation. Note that the level of approximation error (as defined in §A.2) can be measured, so non-anonymous node operators can be penalized for malfeasance. When regulation is not possible, Tâtonnement can be made deterministic by fixing a set of control parameters for each instance and choosing the solution with the lowest approximation error (or lowest unrealized utility, §3.6.2). The Stellar implementation uses a static set of control parameters with one Tâtonnement instance. Node operators could also compete to compute prices accurately, as in [12].

Nondeterministic Overdraft Prevention. SPEEDEX needs to prevent an account from spending more than its balance of an asset. As discussed in §3.3, our implementation considers a proposal valid only if no account is overdrafted after applying the block. This design complicates pipelining of consensus with execution, gives plausible deniability for delaying transactions, and is incompatible with cryptographic commit-reveal schemes.

Instead, given a fixed block of transactions, an implementation could first compute, for each account, the total amount of each asset debited from the account (before applying any credits). If there is any possibility for an account to overdraft in this block, then this amount must exceed the account's balance. As such, to ensure that no accounts overdraft, the implementation can remove all transactions from accounts that might overdraft. Note that this determination is made on a per-account basis, before any transactions are removed, so this filtering requires only one, parallelizable pass over a block of transactions, adding only minimal overhead (§A.9). Furthermore, only accounts that attempt to overdraft are affected.

Other commutativity conflicts, such as cancelling an offer twice or reusing a sequence number, can be handled similarly, by removing all transactions involved in these conflicts. Note that using these filtering criteria, removing a transaction cannot cause a commutativity conflict. The Stellar blockchain plans this approach.

Other Types of Front-Running. The set of pending transactions is public in many blockchains. One might estimate the clearing prices in a future batch and arbitrage the batch against low-latency markets. This could lead to negative externalities (see [102], footnote 1), and could merit combining SPEEDEX with a commit-reveal scheme such as [320, 126]. Such a design requires the deterministic overdraft-prevention scheme above.

Malicious nodes might also delay transactions. An implementation could buffer several blocks of transactions from a consensus protocol into a single SPEEDEX batch. If even one of these consensus blocks is from an honest replica (that does not censor transactions), a user could ensure that their transaction cannot be delayed from one SPEEDEX batch to the next (by broadcasting to all replicas). This requires a consensus protocol with sufficient *chain quality* [175]. Alternatively, some DAG-based protocols [144, 206] simultaneously commit many blocks of transactions from different replicas. Grouping these blocks into one SPEEDEX batch, instead of ordering them arbitrarily, achieves the same censorship-resistance property. These designs would likewise require the deterministic overdraft-prevention scheme.

Linear Program Scalability. The runtime to solve the linear program increases dramatically beyond 60-80 assets, limiting the number of assets in a SPEEDEX batch. A deployment could take advantage of market structure—there are many assets (e.g., stocks) in the real world, but most are linked to one geographic area or economy, and are primarily traded against one currency. We formally show in §A.5 that in this case, the price computation problem can be decomposed between core pricing (i.e., numeraire) currencies and the external stocks. After running Tâtonnement on the core currencies, each stock can be priced on its own relative to a core currency. This lets SPEEDEX support real-world transaction patterns with an arbitrary number of assets and a small number of pricing currencies.

§A.4 points out that setting the commission to 0 simplifies the linear program to one that is more algorithmically tractable at larger numbers of assets. The Stellar implementation uses this version of the linear program.

Limited Trade Types. Trades on SPEEDEX are limited to trades selling a fixed amount of one asset in exchange for as much as possible of another. SPEEDEX does not implement offers to buy a fixed amount of an asset in exchange for as little as possible of another. These buy offers admit the same logarithmic transformation as in §3.5.1, but make the price computation problem PPAD-hard, a complexity class that is widely conjectured to be algorithmically intractable in polynomial time (§A.8). One could compute prices using only sell offers and integrate buy offers in the linear programming step.

Ramseyer et al. [270] show how to integrate Constant Function Market Makers (CFMMs) [69] into the exchange market framework and Tâtonnement. The Stellar implementation uses this integration with its own CFMMs.

3.9 Implementation Details

The standalone SPEEDEX evaluated in §3.6 and §3.7 is a blockchain using HotStuff [316] for consensus. A leader node periodically mints a new block from the memory pool and feeds the block to the consensus algorithm. Other nodes apply the block once it has been finalized by consensus. A faulty node can propose an invalid block. Consensus may finalize invalid blocks, but these blocks have no effect when applied.

The implementation is available open source at `https://github.com/scslab/speedex` and consists of ∼30,000 lines of C++20, plus ∼5,000 lines for our Hotstuff implementation. It uses Intel's TBB library [13] to manage parallel work scheduling, the GNU Linear Programming Kit [237] to solve linear programs, and LMDB [124] to manage data persistence (for crash recovery).

Exchange state is stored in a collection of custom Merkle-Patricia tries; hashable tries allow nodes to efficiently compare state (to check consensus) and build short state proofs.

The rest of this section outlines additional design choices built into SPEEDEX. Additional design choices in §A.11. All optimizations (save §3.9.1) are implemented in the evaluated system.

3.9.1 Blockchain Integration

An existing blockchain with its own (non-commutative) semantics can integrate SPEEDEX by splitting block execution into phases: first applying all SPEEDEX transactions (in parallel), then applying legacy transactions (sequentially). SPEEDEX's scalability lets a blockchain charge only a marginal fee for transactions (to prevent spam). A proof-of-stake integration of SPEEDEX could penalize faulty proposals.

SPEEDEX's economic properties are desirable independent of scalability. The initial Stellar implementation uses two-phase blocks, but the SPEEDEX phase is still implemented sequentially. As a result, the initial implementation is simple (adding only ∼5,000 lines to the server daemon) and the primary benefits are economic. However, because the transaction semantics are commutative, engineers can work to parallelize the implementation as needed, without formally upgrading the protocol (which is more difficult than releasing a software update).

3.9.2 Caches and Tâtonnement

Tâtonnement spends most of its runtime computing demand queries. Each query consists of several binary searches over large lists, so the runtime depends heavily on memory latency and cache performance. Towards the end of Tâtonnement, when the algorithm takes small steps, one query reads almost exactly the same memory locations as the previous query, so the cache miss rate can be extremely low.

Instead of querying the offer tries directly, we precompute for each asset pair a list that records, for each unique limit price, the amount of an asset offered for sale below the price (§A.7). Laying out this information contiguously improves cache performance.

We also execute the binary searches of one Tâtonnement iteration in parallel. One primary thread computes price updates and wakes helper threads. However, each round of Tâtonnement is already fast on one thread—with 50 assets and millions of offers, one round takes 400–600μs. To minimize synchronization latency and avoid letting the kernel migrate threads between cores (which harms cache performance), we operate these helper threads via spinlocks and memory fences. In the tests of §3.6, we see minimal benefit beyond 4–6 helper threads, but this suffices to reduce each query to 50–150μs.

Finally, there is a tradeoff between running more copies of Tâtonnement with different settings and the performance of each copy. More concurrent replicas of Tâtonnement mean more cache traffic and higher cache miss rates.

We accelerate the rest of Tâtonnement by exclusively using fixed-point arithmetic (rather than floating-point).

3.9.3 Batched Trie Design

Our tries use a fan-out of 16 and hash nodes with the 32-byte BLAKE2b cryptographic hash [79]. Both the layout of trie nodes and the work partitioning are designed to avoid having multiple threads writing to the same cache line.

The commutativity of SPEEDEX's semantics opens up an efficient design space for our data structures, which need only materialize state changes once per block. Tries need only recompute a root hash once per block, for example, instead of after every modification. Threads locally build tries recording insertions, which are merged together in one batch operation (which is also parallelizable by redividing local tries into disjoint key ranges). Deletions (when offers are cancelled) are implemented via atomic flags on trie nodes; to enable efficient cleanup of deleted nodes, each node stores the number of deleted nodes beneath it. To facilitate efficient work distribution, each node also stores the number of leaves below it.

SPEEDEX builds in every block an ephemeral trie that logs which accounts are modified; specifically, it maps an account ID to a list of its transactions and to the IDs of transactions from other accounts that modified it. This enables construction of short proofs of account state changes. This trie also uses the same key space as the main account state trie, which lets SPEEDEX use the ephemeral trie to efficiently divide work on the (much larger) account trie.

Memory allocation for an emphemeral trie is trivial because no ephemeral trie node is carried over from one block to the next. Every thread has a local arena, allocation simply increments an arena index, and garbage collection means just setting the index to 0 at the end of a block. We find it to be not a problem if some of the memory in the arena is wasted; we allocate the potential children of an ephemeral trie node contiguously, so a node need only store a 4-byte base pointer (buffer index) and a bitmap denoting the active children. This lets each ephemeral trie node fit in one 64-byte cache line.

3.10 Related Work

Blockchain Scaling. Our approach is inspired by Clements et al. [125], who improve performance in the Linux kernel through commutative syscall semantics.

Chen et al. [120] speculatively execute Ethereum transactions to achieve a ∼6x overall execution speedup. Other approaches to concurrent execution include optimistic concurrency control [179, 311], invalidating conflicting transactions [64], broadcasting conflict resolution information [150, 71], or partitioning transactions into nonconflicting sets [89, 317, 196]. This problem is related to that of building deterministic databases and software transactional memory [264, 309, 294]. Li et al. [226] build a distributed database where some transactions are tagged as commutative.

Empirical work [278, 174] finds that a small number of Ethereum contracts, often token contracts, are historically responsible for the majority of conflicts that limit optimistic execution. A recent Solana [312] outage resulted in part when many transactions conflicted on one orderbook contract [280].

Project Hamilton [234] develops a CBDC payments platform. The authors find that totally-ordered semantics become a performance bottleneck. Unlike SPEEDEX, which stores asset balances in accounts, this system requires the more restrictive unspent transaction output (UTXO) model.

Some systems move transaction execution off-chain, into so-called "Layer-2" networks, each with different capabilities, perfomance, interoperability, and security tradeoffs [263, 203, 18, 10, 262, 11, 35]. Other blockchains [9, 321, 307, 64, 291] split state into concurrently-running shards, at the cost of complicating cross-shard transactions.

(Distributed) Exchanges. Budish et al. [103, 102] argue that exchanges should process orders in batches to combat automated arbitrage and improve liquidity.

Other defenses against front-running include cryptographic commit-reveal schemes [126, 320, 282, 184] or "fair" ordering schemes that assume a bounded fraction of malicious nodes [209, 322, 100, 267]. The front-running attacks that SPEEDEX prevents are not guaranteed to be blocked in these schemes. For example, a replica might plausibly front-run a transaction in [209] by investing in lower-latency network links between itself and other replicas than other replicas have with each other, and commit-reveal schemes do not prevent statistical front-running (guessing the contents of a transaction).

Some blockchains build limit-order DEX mechanisms natively [23, 3] or as smart contracts [19]. Smart contracts known as Automated Market-Makers (AMMs) [55, 194, 157, 241] facilitate passive market-making on-chain [69].

Several exchanges, such as 0x, an upcoming version of DyDx [46], and a past version of Loopring [304, 30] allow settlement on-chain of orders matched off-chain, in pairs or in cycles. Off-chain matching, with only settlement of matched trades on-chain, means users cannot reliably cancel offers (as no cancellation message is ever committed on-chain). StarkEx [21, 91] gives cryptographic tools to prove some correctness properties of an off-chain exchange, but similarly leaves matching logic off-chain [22].

CoWSwap [12, 6] uses mixed-integer programming to clear offers in batches of at most 100 [34]. Solvers compete to produce the best solution. The former Binance DEX [4] computed per-asset-pair prices in each block. The Penumbra DEX uses homomorphic encryption to privately make batch swaps against an AMM, but cannot let users set limit prices [17].

Price Computation. Our algorithms solve instances of the special case of the Arrow-Debreu exchange market [78] where every utility function is linear. Equilibria can be approximated in these markets using combinatorial algorithms such as those of Jain et al. [200] and Devanur et al. [147] and exactly via the ellipsoid method and simultaneous diophantine approximation [199]. Duan et al. [154] construct an exact combinatorial algorithm, which Garg et al. [177] extend to an algorithm with strongly-polynomial running time. Ye [315] gives a path-following interior point method, and Devanur et al. [146] construct a convex program. Codenotti et al. [128, 127] show that a version of the Tâtonnement process [77] converges to an approximate equilibrium in polynomial time. Garg et al. [176] give another algorithm based on demand queries.

3.11 Conclusion

SPEEDEX is a fully on-chain DEX that can scale to more than 200,000 transactions per second with tens of millions of open trade offers. SPEEDEX requires no offchain rollups and no sharding of the exchange's logical state. To make SPEEDEX faster, one can simply give SPEEDEX more CPU cores, without changing the semantics or user interface. Because SPEEDEX operates as a logically-unified platform, instead of a sharded network, SPEEDEX does not fragment liquidity between subsystems and creates no cross-rollup arbitrage.

In addition, SPEEDEX displays several independently useful economic properties. It eliminates risk-free front running; any user who can get their offer to the exchange before a block cutoff time can get the same exchange rate as every other trader. SPEEDEX also eliminates internal arbitrage, which disincentivizes network spam. And finally, SPEEDEX eliminates the need to transact through intermediate, reserve currencies, instead allowing a user to trade directly from one asset to any other asset listed on the exchange, with the same or better market liquidity as the trader would have gotten by trading through a series of intermediate currencies.

SPEEDEX is free software, available at `https://github.com/scslab/speedex`.

Chapter 4

Fair Ordering via Streaming Social Choice Theory

4.1 Introduction

A core problem when building distributed systems is in ordering transactions on said system. Replicated state machine architectures [257, 217, 316, 255] typically first comes to consensus on a totally ordered *log* of *transactions*, and then apply each transaction in the log to modify a *state machine*. We study a model where n *replicas* operate a copy of a state machine, and receive transactions from a set of *clients*.

However, replicas may receive transactions in different orderings. Different local views on transaction order must be aggregated into a single, total order.

This problem is similar to the classic problem of social choice theory [188, 232, 132, 133, 76], where an election organizer computes an ordering over a finite set of candidates given a ranking of those candidates from each of n voters. The "preferred ranking" of a replica is the order in which it receives transactions. There is a rich literature on the properties of different voting rules, but two features make our setting novel and distinct.

First, the number of transactions in the system is unbounded; as time passes, clients send more and more transactions. In the language of the classical literature, the voting rule must aggregate rankings (where each ranking is a total-ordering, isomorphic to ω) over a countably infinite set of candidates. Such a ranking must be well-defined. For example, the Copeland method [136] is not well-defined, as each candidate pairwise defeats a countably-infinite set of candidates—but this challenge can be addressed by instead ranking by to minimizing the number of pairwise losses. We focus on the Ranked Pairs method, which is defined by an iterative process— but such a process does not terminate on an infinite input.

And second, rankings are not fully known at any finite time, but arrive in a streaming fashion (as replicas receive transactions). Nevertheless, the system must progress, which means adding transactions to the output log with incomplete information. Replicas apply a transaction to local copies of the state machine as soon as it is in the output log (and, in a real-world deployment, downstream systems might make decisions based on transactions being committed to the log), so transactions in the output log cannot be rearranged later. Similarly, transactions can only be appended to the end of the log. Clients would also like to minimize the time between when they send a transaction and when it is added to the log.

An additional complication is that some replicas might misrepresent the order in which they received transactions over the network. One desirable property of an aggregation rule is that the influence of a (colluding) subset of replicas is bounded and precisely quantified. While we think this streaming problem with only the above two considerations is interesting in its own right, adversarial manipulation of transaction ordering poses an additional challenge. Many public blockchains use this architecture [249, 308, 242, 274], but allow widespread ordering manipulation [143] so as to e.g. "front-run" trades on an exchange, extracting risk-free profit at another user's expense.

4.1.1 Our Results

Prior work [209, 207, 208] study this problem of aggregating orderings "fairly," but to the best of our knowledge, this work is the first to directly pose the question as a streaming version of the classic social choice problem. For a ranking system to be well-defined on a countably infinite set, we show that it suffices for there to exist a "monotonic" and "asymptotically live" algorithm, which does not undo a decision when its (finite) input is extended and eventually outputs every transaction. The challenge in building such an algorithm is in determining when an algorithm has enough information to make a decision that is consistent with its hypothetical output on any extension of its input orderings.

We focus here on the ordering known as Ranked Pairs [295]. Analysis of the graph of ordering dependencies gives an important structural lemma about what pieces of information Ranked Pairs uses to determine its output. Using this lemma, we can carefully track how uncertainty about as-yet-unseen votes on the ordering between transactions propagates throughout Ranked Pairs's operation and instantiate Ranked Pairs in our streaming setting (§4.6).

Ranked Pairs iterates over a directed graph in order of the weight of each edge. We find that the tiebreaking rule that determines the ordering of edges of equal weight is a critical part of the algorithm. Using a naive deterministic rule, our streaming algorithm might never output any transactions. We then show in §4.7 that careful manipulation of the tiebreaking (in essence, dynamically generating a tiebreaking rule) guarantees asymptotic liveness.

"Fair" in prior work means γ-batch-order-fairness. The output ordering is divided into batches, and while transactions within a batch may be ordered arbitrarily, if γn replicas receive a transaction

tx before another transaction tx', then tx cannot be in a later batch than tx'. We first strengthen in §4.4 this definition to precisely captures the intuition that batches should be *minimal.* Minimality is essential to a useful definition fairness, as otherwise, a protocol vacuously satisfies the definition by placing all transactions in the same batch. Minimality must be defined carefully in the presence of faulty replicas; we show that γ-batch-order-fairness cannot be satisfied simultaneously with exact notions of "minimalily" of batch size and faulty replicas.

We call our definition (γ, δ)-minimal-batch-order-fairness. Simply put, if a γ fraction of replicas vote for tx before tx', then the output ordering should include tx before tx' unless there is a sufficiently strong reason to put tx' before tx—that is, a sequence of transactions $(tx_1, tx_2, \dots, tx_{k-1}, tx_k)$, with $tx_1 = tx$ and $tx_k = tx'$, where at least a $\gamma - 2\delta$ fraction of replicas vote for tx_i before tx_{i+1} for $1 \leq i \leq k-1$. As an example, given f faulty replicas out of n total replicas and a fixed γ, prior work [209] satisfies $(\gamma, \frac{f}{n})$-minimal-batch-batch-order-fairness.

We then show in §4.5 that Ranked Pairs voting satisfies our fairness criterion. Importantly, unlike all prior work, our algorithm does not require knowledge of or any bound on the number of faulty replicas [1] or a fixed choice of γ. Instead, our protocol satisfies $(\gamma, \frac{f}{n})$-minimal-batch-order-fairness for every γ and every $\frac{f}{n}$ simultaneously. Fairness guarantees smoothly weaken as f increases.

Finally, a synchronous network assumption combined with our tiebreaking manipulation guarantees that our algorithm outputs each transaction after a finite amount of time. If at most Δ time elapses between when a transaction tx is sent and when every honest replica votes on it, for example, our algorithm outputs tx after a delay of at most $O(n\Delta)$. This can be reduced to $O(\Delta)$, by rounding edge weights— although rounding each fraction of replicas who vote for one transaction before another to the nearest $\frac{1}{k}$ reduces the fairness guarantee to $(\gamma, \frac{f}{n} + \frac{1}{2k})$-minimal-batch-order-fairness (again for every γ simultaneously, and any f).

An explicit instantiation of this protocol would require a reliable, fault-tolerant communication layer to allow replicas to communicate their ordering votes with each other. We keep this separation explicit. Any consensus protocol will satisfy our requirements, although precise results h each other. about liveness (§4.7.2) naturally depend on the choice of protocol and network assumptions. §4.8 gives an example of an explicit instantiation.

4.2 Preliminaries and System Model

We consider a model in which there are n *replicas* cooperating to develop a total ordering of transactions. We number replicas from 1 to n. Transactions are received over the network from *clients.* The ordering in which a replica receives transactions is that replica's observed ordering.

[1] An instantiation requires a fault-tolerant way for replicas to broadcast their ordering votes. Such protocols may require a bound on the number of faulty participants, but these are different kinds of faults in a different part of the system.

Definition 4.2.1 (Ordering Preference)**.** *An Ordering Preference on a (finite or countably-infinite) set of transactions is a total ordering* [2] $\hat{\sigma_i} = (tx_{i_1} \preccurlyeq tx_{i_2} \preccurlyeq tx_{i_3} \preccurlyeq \dots)$

However, at any finite time, each replica can have received only a finite number of transactions. Replicas periodically submit these finite "ordering votes" to a ranking algorithm.

Definition 4.2.2 (Ordering Vote)**.** *A replica i submits to a ranking algorithm an ordering vote on a set of k transactions, $\sigma = (tx_{i_1}, ...tx_{i_k})$. (where $tx_{i_{j_1}} \preccurlyeq tx_{i_{j_2}}$ for $j_1 < j_2$).*

We say that σ' extends σ if σ is an initial segment of σ' (as a convention, σ extends itself).

Definition 4.2.3 (Ranking Algorithm)**.** *A deterministic ranking algorithm $\mathcal{A}(\sigma_1, \dots, \sigma_n)$ takes as input an ordering vote from each replica and outputs an ordering σ on a subset of the transactions in its input.*

The output need not include every transaction in the input.

We say that a replica is *honest* if its true, observed ordering always extends its ordering vote, and if whenever it submits a vote, it contains all transactions that the replica has observed at that time. Otherwise, the replica is *faulty.*

We assume that at most f of the n replicas are faulty, although our algorithms do not require knowledge of f. Faulty replicas might fail to vote on a transaction, but §4.8 addresses this difficulty. Therefore, in the rest of this work, we assume that every replica eventually votes on every transaction.

4.3 Related Work

Aequitas [209] and Themis [208] propose the notion of γ-batch-order-fairness and instantiate ordering protocols based on aggregating ordering votes. Cachin et al. [110] propose an equivalent definition κ-differential-order-fairness (Theorem D.1, [208]). Our work strengthens this definition to (γ, δ)-minimal-batch-order-fairness. §4.4 discusses our work in relation to this line of work, and specifically discuss the shortcomings of γ-batch-order-fairness and the difference between it and (γ, δ)-minimal-batch-order-fairness. Zhang et al. [322] sort by median reported timestamp, for a linearizability guarantee incomparable with our definition.

Concurrently with this work, Vafadar and Khabbazian [298] observe that extra transactions that create Condorcet cycles can bypass the constraints γ-batch-order-fairness puts on the output (using an example akin to Example B.2.1 and Lemma 4.4.8). Arrow's impossibility theorem shows no non-dictatorial, unanimous method is immune to manipulation via sending extra transactions [76].

Vafadar and Khabbazian also propose using Ranked Pairs to sort transactions within the batches output in Themis. However, no modification to the ordering of transactions within a batch output in

[2]Isomorphic to a subset of ω

Aequitas can give the Ranked Pairs ordering, because Ranked Pairs might need to interleave batches that Aequitas considers incomparable (as demonstrated in §B.4). As discussed in §4.4.3, the batches in Themis are defined slightly differently than in Definition 4.4.1 and in Aequitas, because Themis, when computing a graph of ordering dependencies, often ignores its choice of γ. In so doing, it merges together two batches unless, for any transaction tx in one and tx' in the other, a majority of replicas vote $tx \preccurlyeq tx'$ (bypassing the counterexample of §B.4).

However, Themis's round-based protocol presents a different problem insurmountable by any such process that sorts its batches separately. Themis may finalize the composition of a batch before it has enough information to ensure that, according to the Ranked Pairs ordering, all transactions in the batch come before all those not in the batch (and not already output). This means that Themis may output two batches in sequence where running Ranked Pairs on each batch separately produces a different output than running Ranked Pairs on the union of the two batches together. §B.3 gives an explicit example.

Classical social choice studies the problem of ordering a finite set of candidates given a complete set of preferences from each voter [76]. Prior work studies related models in a limited-information setting. Lu and Boutillier [235], Ackerman et al. [54], and Cullinan et al. [141] study the setting where a social choice rule has incomplete access (a partial ordering) to a voter's preferences over a finite set of candidates. Conitzer and Sandholm [134] study the communication complexity of a variety of voting rules. Fain et al. [163] study a randomized rule that uses only a constant number of queries to voter preferences.

Fishburn [168] shows that Arrow's impossibility theorem does not hold when the set of voters is infinite, although Kirman and Sondermann [211] find "dictatorial sets" of voters. Grafe and Grafe [186] extend these results to the case of infinitely many alternatives, subject to a continuity condition on the space of ranking preferences. Chichilnisky and Heal [123] and Efimov and Koshevoy [156] study the rules admissible in the infinite voters setting, given a topology on ranking preferences.

Crisman et al. [140] study the ordering problem where the output must be a cyclic permutation.

Some systems commit to an ordering before revealing transaction contents, using threshold encryption or commit-reveal schemes [238, 126, 320]. These ideas could be used in conjunction with a streaming ordering algorithm.

Kavousi et al. [205] randomly shuffle blocks of encrypted transactions. Some systems [269, 6, 17] process transactions in unordered batches, eliminating the need to totally order them. Kelkar et al. [207] instantiate Aequitas when the number of replicas is unknown and not fixed. Ferreira and Parkes [167] construct an application-specific ordering. Mamageishvili et al. [239] allow payments to prioritize transactions, using the median-timestamp mechanism of [322]. Li et al. [228] verifiably compute orderings in hardware enclaves [138].

4.4 Defining Fair Ordering

4.4.1 Batch Order Fairness

Our discussion here follows that of [209, 208, 110], but gives, unlike prior work, a definition that is meaningful and achievable.

Honest replicas might receive transactions over the network in a different order. To resolve discrepancies "fairly", To resolve disagreements "fairly," recent work [209] proposes a notion known as "γ-Batch-Order-Fairness," which we reproduce below from Definition III.1 of [208] (a more complicated but functionally similar definition is used in [209]).

Definition 4.4.1 (γ-batch-order-fairness)**.** *Suppose that tx and tx' are received by all nodes. If γn nodes received tx before tx', then a ranking algorithm outputs tx no later than tx'.*

Aequitas [209] outputs transactions in batches, and "no later than" means "in the same batch." Transactions within a batch are ordered arbitrarily. Themis [208] outputs a total ordering, with the assertion that this total ordering could be divided into contiguous (disjoint) batches satisfying this property.

One challenge is the parameter γ. It is not obvious if a higher or lower γ gives stronger fairness properties. Indeed, there are simple examples (with no faulty replicas) where Definition 4.4.1 (actually the stronger Definition 4.4.5) only implies ordering restrictions at high values of γ, others at only low values, and still others at only intermediate values (§B.2). Prior work relies on a fixed choice of γ, but our algorithms satisfy Definition 4.4.5 for all values of γ simultaneously.

4.4.2 Batch Minimality

Unfortunately, Definition 4.4.1 is insufficient to provide meaningful ordering guarantees. A protocol that outputs all transactions in the same batch vacuously satisfies this definition, and if transactions within a batch can be ordered arbitrarily, then every ordering satisfies Definition 4.4.1 for every γ.

Prior work [208] recognizes the need for "minimality".However, the notion of minimality in prior work is not achievable in the presence of faulty replicas (Theorem 4.4.4). Towards a feasible notion of minimality, we first define a graph denoting ordering dependencies.

Definition 4.4.2 (Ordering Graph)**.** *Suppose that replicas report orderings $(\sigma_1, ..., \sigma_n)$ on a set of transactions V. The Ordering Graph $G(\sigma_1, ...\sigma_n)$ is a complete, weighted directed graph with vertex set V and, for each transaction pair (tx, tx'), an edge of weight $w(tx, tx') = \alpha$ if αn replicas report receiving tx before tx'.*

Any notion of "minimal" should at least require that when there are no cycles in ordering dependencies, the output is consistent with every ordering dependency (as in Theorem IV.1 part 1, Themis [208]). Exact γ-minimality generalizes this notion to the case where cycles in ordering dependencies exist.

Definition 4.4.3 (Exact γ-minimality)**.** *For any pair of transactions tx, tx', if γn replicas receive $tx \preccurlyeq tx'$ but tx' is output before tx, then there exists a sequence of transactions $tx' = tx_1, ..., tx_k = tx$ where γn replicas receive $tx_i \preccurlyeq tx_{i+1}$ and tx_i is output before tx_{i+1} for all i.*

Unfortunately, this notion of minimality is *impossible* to achieve (for any γ) in the face of *any* faulty replicas. The condition on f in Theorem 4.4.4 is a technical artifact of a construction in the proof (related to the pigeonhole principle). For some parameter settings, such as $n = 7$ with $\gamma = \frac{2}{3}$, the condition admits $f = 1$.

Theorem 4.4.4. *No protocol can achieve γ-batch-order-fairness with exactly γ-minimal output batches for any n and f greater than $n \mod (n - \lceil \gamma n \rceil + 1)$ (for $\gamma > 1/2$).*

In fact, the construction in the proof contains no cycles in the ordering dependency graph of weight γ, thereby applying even to a weaker version of Definition 4.4.3 that applies only to the case where the ordering graph, restricted to edges of weight at least γ, has no cycles.

Proof. Define m=$\lceil \gamma n \rceil$, and $k = \lfloor \frac{n}{n-m+1} \rfloor$. Consider the process of choosing an ordering between k transactions $tx_1, tx_2, ..., tx_k$. Assume that every node submits a vote on transaction ordering (i.e. the theorem holds even when faulty nodes are required to submit a vote).

An ordering dependency arises, therefore, if and only if there are at least m votes for some tx before another tx'.

Divide the set of nodes into disjoint groups G_i for $1 \leq i \leq k$ of size $n - m + 1$. The remainder are the faulty nodes.

Suppose that nodes in group i receive transactions in order $tx_i \preccurlyeq ... \preccurlyeq tx_k \preccurlyeq tx_1 \preccurlyeq ... \preccurlyeq tx_{i-1}$, and that the faulty nodes receive the same ordering of transactions as the nodes in group G_1.

As such, the transaction pairs tx_i, tx_j that receive at least m votes are exactly those with $i < j$, and there are no cycles in this ordering dependency graph. Thus, any protocol must output the ordering $tx_1, ..., tx_k$.

However, for any protocol that lacks knowledge of which nodes are faulty, this scenario is indistinguishable from one where the faulty nodes had received transactions in the ordering received by group G_2, but reported the ordering observed in G_1. As such, the protocol would necessarily output the ordering observed by G_1.

However, in this case, the only correct output would have been $tx_2, ..., tx_k, tx_1$, as only the transaction pairs tx_i, tx_j with $i < j$ for $i \neq 1$ or $j = 1$ received m votes for $tx_i \preccurlyeq tx_j$. □

The core difficulty is that faulty replicas can cause the weight on an edge observed by any algorithm to be different from the true, ground truth weight. Intuitively, if a protocol has a reliable communication layer (i.e. every honest replica is able to submit a vote on its ordering preference), then the worst that a faulty replica can do is to misreport its ordering preferences. If there are f faulty replicas, then the faulty nodes can artificially reduce or increase the fraction of replicas that vote for $tx \preccurlyeq tx'$ by at most $\frac{f}{n}$. As a protocol cannot distinguish a faulty replica from an honest one by its votes, we therefore must take some error in edge weights into account in our definition of minimality.

Definition 4.4.5 ((γ, δ)-minimal-batch-order-fairness)**.** *An ordering is* (γ, δ)*-minimally-batch-order-fair if, for any transaction pair* (tx, tx') *that is received in that order by at least* γn *replicas but output by the protocol in the reverse ordering, then there is a sequence of transactions* $tx' = tx_1, ..., tx_k = tx$ *where at least* $(\gamma - 2\delta)n$ *replicas receive* $tx_i \preccurlyeq tx_{i+1}$ *and* tx_i *is output before* tx_{i+1}*.*

Definition 4.4.5 captures the notion that a protocol cannot distinguish between a δ-fraction of replicas misreporting a transaction ordering. Given this indistinguishable δ fraction, the protocol outputs minimally-sized batches. Definition 4.4.3 corresponds to the case of $\delta = 0$.

This definition does not explicitly discuss "batches" or "minimality," but approximately minimal batches (the strongly connected components of Lemma 4.4.6) can be recovered from any ordering satisfying it. The second condition of Lemma 4.4.6 limits the size of output batches.

Lemma 4.4.6. *Suppose that an output ordering satisfies* (γ, δ)*-minimal-batch-order-fairness. Compute the ordering graph* $\hat{G}$*, drop all edges with weight below* $\gamma - \delta$*, and compute the strongly connected components of the remainder.*

- *If* γn *replicas received* $tx \preccurlyeq tx'$*, then either* tx *and* tx' *are in the same strongly connected component, or all transactions in the component containing* tx *are output before any transactions in the component containing* tx'*.*

- *If* γn *replicas receive* $tx \preccurlyeq tx'$ *and there is no sequence of transactions* $tx' = tx_1, ..., tx_k = tx$ *where at least* $(\gamma - 2\delta)n$ *replicas receive* $tx_i \preccurlyeq tx_{i+1}$*, then all transactions in the component containing* tx *are output before any transactions in the component containing* tx'*.*

Proof. If at least γn replicas receive $tx \preccurlyeq tx'$, then the edge (tx, tx') is included in the thresholded ordering graph, and if less than $(\gamma - 2\delta)$ replicas receive $tx \preccurlyeq tx'$, then the edge is not included.

Thus, if at least γn replicas receive $tx \preccurlyeq tx'$, then either tx and tx' are in the same strongly connected component, or there is an edge from the component containing tx to that containing tx'. And if, additionally, there is no sequence of transactions from tx' to tx as in the lemma statement, then tx and tx' must be in different components.

If there is an edge from one strongly connected component to another, then all transactions in the first component must be output before any in the second (or else a violation of Definition 4.4.5) would occur. $\square$

There is one subtle difference between this definition and (an analogue of) the notion of "minimality" in Theorem IV.1 of Themis, and the sequence of batches output in Aequitas. When there are two disjoint strongly connected components with no dependencies (of strength at least γ) from one to the other, Definition 4.4.5 allows the output to interleave these components. This may be strictly required (§B.4).

4.4.3 Comparison to Prior Work

Before constructing our protocol, we first apply Definition 4.4.5 to related prior work.

Lemma 4.4.7. *Aequitas with parameters (γ, f) achieves $(\gamma, \frac{f}{n})$-minimal-batch-order-fairness.*

Note that Aequitas requires $\gamma - \frac{1}{2} > \frac{2f}{n}$. Lemma 4.4.7 is a stronger version of Theorem 6.6 of [209].

Proof. Aequitas considers a transaction pair (tx, tx') as an ordering dependency if at least $\gamma n - f$ report receiving tx before tx' (part 3.b.ii of §6.2 of [209]). This ensures that the protocol considers every ordering dependency that truly receives at least γn votes, and might include any edge that receives at least $\gamma n - 2f$ votes.

Aequitas sequentially outputs minimal cycles in its computed graph of ordering dependencies (not Definition 4.4.2), ordering within each cycle arbitrarily. This means that if some pair that receives at least γn votes is output in reversed order, then it must be part of a cycle of dependencies, and each of these dependencies must have received at least $\gamma n - 2f$ votes in order to be included in Aequitas's computed graph. □

By contrast, Ranked Pairs (Theorem 4.5.1) achieves $(\gamma, \frac{f}{n})$-minimal-batch-fairness for every γ simultaneously, and for any f (our work does not require knowledge of a bound on f). Aequitas also is not asymptotically live (Definition 4.6.2).

Themis claims (Theorem IV.1, [208]) to output a sequence that can be partitioned into minimal batches, which appears to contradict Theorem 4.4.4. However, Themis relaxes its notion of an "ordering dependency".

Where γ-batch-order-fairness would create a dependency for tx not after tx' if at least γn replicas receive tx before tx', the relaxation implied by Definition III.1 of [208] creates such a dependency if at least $n(1-\gamma)+1$ receive tx before tx' (that is, it cannot be the case that more than γn receive tx' before tx). Under this definition, the construction in Theorem 4.4.4 constitutes what the authors of [208] call a "weak Concorcet Cycle".

Extra ordering dependencies, however, can artificially merge batches together, giving counterintuitive results.

Lemma 4.4.8. *In Themis [208], even if all nodes report tx before tx' and there are no faulty replicas nor cycles in ordering dependencies (as defined in Definition 4.4.1) for any $\gamma > 1/2$, the output may put tx' before tx.*

Lemma 4.4.8 follows from the fact that when two transactions tx, tx' receive $n - f$ votes in one round of Themis, it determines whether to consider as a dependency $tx \preccurlyeq tx'$ or $tx' \preccurlyeq tx$ by majority vote (disregarding γ). The minimality statement of Theorem IV.1 of [208] is with regard to this weaker notion of dependency.

Proof. Suppose that the number of nodes is even, and break the nodes into two groups. Group one receives transactions $tx_1 \preccurlyeq tx_2 \preccurlyeq tx_3$, while group two receives transactions $tx_3 \preccurlyeq tx_1 \preccurlyeq tx_2$.

All nodes submit their local orderings to the protocol correctly. Themis proceeds in rounds; the round under consideration consists only of these three transactions.

Themis builds a graph of what it considers ordering dependencies on the transactions in a round (Figure 1, part 1, [208]). Let $w(tx, tx')$ be the number of replicas that vote for tx before tx'.

An ordering dependency between tx and tx' is included if (1) $w(tx, tx') \geq w(tx', tx)$ and (2) $w(tx, tx') \geq n(1 - \gamma) + f + 1$. Themis assumes $n > 4f/(2\gamma - 1)$, or equivalently, $2f < n(\gamma - 1/2)$, so $(1 - \gamma) + (f + 1)/n \leq (1 - \gamma) + (2f)/n \leq (1 - \gamma) + (\gamma - 1/2) \leq 1/2$. Ties are broken in an unspecified (deterministic) manner.

As such, (depending on the tiebreaking), Themis may consider as valid ordering dependencies the pairs (tx_1, tx_2), (tx_2, tx_3), and (tx_3, tx_1). Within a strongly connected component of its dependency graph, Themis computes a Hamiltonian cycle, then outputs transactions by walking arbitrarily along that cycle. As such, a possible output of Themis is the ordering $tx_2 \preccurlyeq tx_3 \preccurlyeq tx_1$.

Thus, all nodes received tx_1 before tx_2, and there were no cycles of ordering dependencies for any $\gamma > 1/2$, but the protocol could output tx_2 before tx_1.

Note that Themis only waits for $n - f$ votes from replicas before proceeding, not the full n. This difference is immaterial if $f = 0$. However, the construction above works only if the numbers of votes for $tx_3 \preccurlyeq tx_1$ and for $tx_2 \preccurlyeq tx_3$ are both at least $n(1 - \gamma) + (f + 1)$. Tighter analysis shows that $n(1-\gamma)+(f+1) = n(1-\gamma)+(2f)-(f-1) \leq n(1-\gamma)+n(\gamma-1/2)-(f-1) = n/2-(f-1) < (n-f)/2$, so the construction still works.

For γ larger than $1/2 + \varepsilon$ for some small ε and sufficiently large n, this construction could be repeated with more transactions in the cycle. This eliminates the need to abuse a tiebreaking rule when votes are evenly divided. □

4.5 Ranked Pairs Voting

We turn now to the Ranked Pairs method [295]. Ties in edge weights are broken arbitrarily.

Theorem 4.5.1. *Given a ordering vote (on every transaction in a finite set) from every replica, Ranked Pairs Voting simultaneously satisfies $(\gamma, \frac{f}{n})$-minimal-batch-order-fairness for every γ, and does not depend on any fixed bound on f.*

ALGORITHM 1: Ranked Pairs Voting

```
Input: An ordering vote from each replica $\{\sigma_i\}_{i\in[n]}$
$G = (V, E, w) \leftarrow G(\sigma_1, ..., \sigma_n)$ ;                /* Compute ordering graph */
$H \leftarrow (V, \emptyset)$ ;                  /* Initialize empty graph on same vertex set */
Sort $E$ by edge weight
foreach $e \in E$ do
    Add $e$ to $H$ if it would not create a directed cycle in $H$
end
Output: The topological sort of $H$
;  /* The loop considered every edge, so the topological sort of $H$ is unique
 */
```

Lemma 4.4.6 and Theorem 4.5.1 together imply an important property. The output of Ranked Pairs can be divided into batches consistent with γ-batch-order-fairness for *any* γ, not just a single γ (subject to the interleaving discussed in §4.4.2). Unlike prior work, there is no need to choose a γ or order transactions within a batch.

Proof. Ranked Pairs Voting ensures that if some edge of weight γ is not included in the output graph G — that is to say, if γn nodes vote for tx before tx', but the output ordering has $tx' \preccurlyeq tx$ — then there must be a directed path of edges in G already from tx' to tx. As the algorithm looked at these edges before the current edge, these edges must have weight (as observed by the algorithm) at least γ.

Note that if there are f faulty nodes, these nodes can, at most, adjust the weight (observed by the algorithm) on any particular transaction pair by at most $\frac{f}{n}$.

Thus, an edge with true weight γ is only reversed if there exists a path in the ordering graph (that is included in G) in the opposite direction of minimum true weight at least $\gamma - \frac{2f}{n}$.

Nowhere in the algorithm do its choices depend on γ or f (or n). □

4.6 Streaming Ranked Pairs

We now turn to the streaming setting that is the focus of our work. There may be an infinite set of transactions to consider overall, but at any finite time, an algorithm must compute an ordering on a subset of the transactions that have been seen thus far. In a real-world system, such an algorithm would be run with some periodic frequency, and replicas would periodically append to their ordering votes.

Our algorithms must be *monotonic* (Definition 4.6.1)—if replicas extend their ordering votes, the algorithm, when run again on the extensions, can only extend its prior output.

Definition 4.6.1 (Monotonicity)**.** *A sequencing algorithm $\mathcal{A}(\cdot, \dots, \cdot)$ is monotonic if, given two sets of ordering votes $(\sigma_1, \dots, \sigma_n)$ and $(\sigma'_1, \dots, \sigma'_n)$, such that σ'_i extends σ_i for all $i \in [n]$, $\mathcal{A}(\sigma'_1, \dots, \sigma'_n)$*

extends $\mathcal{A}(\sigma_1, \ldots, \sigma_n)$.

Observe that if a sequencing algorithm is monotonic, it implies a well-formed definition for aggregating a set of orderings on countably infinite sets of transactions, not just on finite inputs. Specifically, tx comes before tx' in the overall (infinite) case if there exists a finite subset of the input orderings such that the algorithm puts $tx \preccurlyeq tx'$ in its output.

A non-vacuous condition is also required—the algorithm that never outputs anything is monotonic.

Definition 4.6.2 (Asymptotic Liveness). *A sequencing algorithm $\mathcal{A}(\cdot, \ldots, \cdot)$ is asymptotically live if, given any set of countably infinite ordering votes $(\hat{\sigma_1}, \ldots, \hat{\sigma_n})$ and any transaction tx in those votes, there exists an N such that when each $\hat{\sigma_i}$ is trunctated to the first N elements of the ordering to produce σ_i, tx is included in $\mathcal{A}(\sigma_1, \ldots, \sigma_n)$.*

Corollary 4.6.9 shows that our version of Ranked Pairs, Algorithm 2, is monotonic. However, Algorithm 2 is not asymptotically live (§B.5). Theorem 4.7.5 shows that a modification to the tiebreaking rule between edges of equal weight, as in Algorithm 3, produces an asymptotically live algorithm.

4.6.1 Decisions Are Local

Implementing Ranked Pairs voting in this streaming setting requires first understanding when the algorithm chooses or rejects an edge in the ordering graph.

Lemma 4.6.3. *If Ranked Pairs rejects an edge (tx, tx') of weight α, then it must choose some directed path of edges from tx' to tx, where each edge has weight at least α.*

Proof. If Ranked Pairs rejects an edge (tx, tx') of weight α, then adding the edge would have created a cycle among the edges the algorithm had already chosen. Ranked Pairs looks at edges in order of weight, so all of the edges on the already chosen path from tx' to tx must have weight at least α. □

Observation 4.6.4. *Ranked Pairs chooses every edge of weight 1.*

Proof. There cannot be any cycles of edges, all of weight 1. Such a situation would require every replica to submit a cycle of preferences, which is impossible, given that each ordering vote σ is a total, linear order. □

Furthermore, the set of vertices that such a path can visit are bounded to a local neighborhood of tx_A and tx_B.

Definition 4.6.5. *Let tx be any transaction.*

1. *Let P_{tx} be the set of all preceeding transactions; that is, all tx' with $w(tx', tx) = 1$.*
2. *Let Q_{tx} be the set of all concurrent transactions; that is, all tx' with $0 < w(tx', tx) < 1$.*
3. *Let R_{tx} be the set of all subsequent transactions; that is, all tx' with $w(tx', tx) = 0$.*

Lemma 4.6.6. *If ranked pairs chooses all edges in a path from tx_B to tx_A, no transactions on the path are in R_{tx_A} or P_{tx_B}.*

Proof. If the path visits a transaction tx in R_{tx_A}, then it creates a cycle with the edge from tx_A to tx (which ranked pairs must choose, by Observation 4.6.4). If the path visits a transaction tx in P_{tx_B}, then it creates a cycle with the edge from tx to tx_B. □

Lemma 4.6.6 lets the streaming algorithm decide whether to choose an edge given only a finite amount of information. Once the algorithm sees a vote for a transaction tx from every replica, it can compute the weight in the ordering graph for every edge (tx, tx') and (tx', tx). Unseen transactions must be in R_{tx}.

Even so, a streaming algorithm cannot always know whether or not Ranked Pairs would choose an edge. As such, we allow our algorithms to leave an edge in an "*indeterminate*" state, and design our algorithms to account for indeterminate edges when considering subsequent edges.

First, we construct an ordering graph that captures the information available to an algorithm at a finite time.

Definition 4.6.7 (Streamed Ordering Graph)**.**
Suppose that each replica submits an ordering vote $(\sigma_1, ..., \sigma_n)$. Let V be the set of all transactions that appear in each vote, and let $\hat{v}$ be the "future" vertex.

The Streamed Ordering Graph is a weighted, directed, complete graph $\hat{G} = (V', E)$ on vertex set $V \cup \{\hat{v}\}$.

For each tx and tx', set weights $w(tx, tx')$ and $w(tx', tx)$ as in Definition 4.4.2. Set $w(tx, \hat{v}) = 1$ for all tx. If there exists tx' that appears in the votes of some but not all replicas and which preceeds tx in at least one replica's vote, set $w(\hat{v}, tx) = 1$, and otherwise $w(\hat{v}, tx) = 0$.

Conceptually, the "future" vertex $\hat{v}$ represents all transactions that have not received votes from all replicas. Implicitly, all edge weights not computable from the available information are upper-bounded by 1. An implementation might compute a tighter upper bound on the weights of edges $(\hat{v}, tx)$. Subsequent arguments require only that the assigned weight is an upper bound of the true weight.

4.6.2 A Streaming Algorithm

We can now construct an algorithm to compute Ranked Pairs in our streaming setting. Instead of either including or rejecting every edge, Algorithm 2 has the option to declare an edge as *indeterminate*; that is, the algorithm is not able to determine whether to include or exclude the edge with the information in its input.

ALGORITHM 2: Streaming Ranked Pairs

```
Input: An ordering vote from each replica $\{\sigma_i\}_{i\in[n]}$
$\hat{G} = (V, E, w) \leftarrow \hat{G}(\sigma_1, ..., \sigma_n)$ ;                /* Compute streamed ordering graph */
$H \leftarrow (V, \emptyset)$ ;                /* Initialize empty graph on same vertex set */
foreach edge $(tx, \hat{v})$ and $(\hat{v}, tx)$ with $w(e) = 1$ do
    Add $e$ to $H$, and mark it as indeterminate
end
Sort $E$ by edge weight, with a fixed tiebreaking rule.
foreach $e = (tx_i, tx_j) \in E$ do
    $U_{tx_i,tx_j} \leftarrow (V \setminus (R_{tx_i} \cup P_{tx_j})) \cup \{\hat{v}\}$
    ;   /* Recall that $\hat{v}$ is the "future" vertex of the streamed ordering graph
     */
    If there is a directed path in $H$ restricted to $U_{tx_i,tx_j}$ from $tx_j$ to $tx_i$ where every edge is
      determinate, then do not include the edge in $H$.
    If there is no directed path in $H$ restricted to $U_{tx_i,tx_j}$ from $tx_j$ to $tx_i$ where every edge is
      either determinate or indeterminate, add $(tx_i, tx_j)$ to $H$ and mark it as determinate.
    Otherwise, add $(tx_i, tx_j)$ to $H$ and mark it as indeterminate.
end
Output: The topological sort of $H$ (with sorting comparisons restricted to determinate
        edges), up to but not including the first transaction $tx$ bordering an
        indeterminate edge
```

To prove correctness of this algorithm, it suffices to show that the output of the algorithm is monotonicand that the output is consistent with the non-streaming version of Ranked Pairs. By consistency, we specifically mean that at any point, if clients were to stop sending new transactions and all replicas eventually voted on every replica, Algorithms 1 and 2 would produce the same output.

Lemma 4.6.8. *Consider a counterfactual scenario where clients stop sending transactions, all replicas eventually receive every transaction, and every replica includes every transaction in its output vote.*

Whenever Algorithm 2 includes a determinate edge in H (resp. excludes an edge), that edge is included (resp. excluded) in the output of the non-streaming Ranked Pairs on the counterfactual input.

Proof. Suppose that the lemma statement holds for all edges earlier in the Ranked Pairs ordering. Specifically, assume that Ranked Pairs would choose every higher *determinate* edge, not choose any higher rejected edge, and might or might not choose any higher *indeterminate* edge (and that all edges of higher weight are all either already rejected, chosen as *determinate*, or chosen as *indeterminate*).

By Lemma 4.6.6, Ranked Pairs would choose (tx_i, tx_j) if and only if there does not exist a path of already chosen edges higher in the ordering from tx_j to tx_i that does not enter R_{tx_i} or in P_{tx_j}.

If there exists a path of *determinate* edges in U_{tx_i,tx_j}, then Streaming Ranked Pairs rejects (tx_i, tx_j), and if there does not exist a path of *determinate* or *indeterminate* edges, then Streaming Ranked Pairs accepts (tx_i, tx_j). Otherwise, the edge is left *indeterminate*.

Note that Streaming Ranked Pairs considers edges in the same ordering that Ranked Pairs would (relative to the restricted set considered). However, at any point, due to the initialization of H with *indeterminate* edges, the set of edges in which the algorithm searches for a cycle might include more edges than those that would be considered by Ranked Pairs. However, these extra edges are *indeterminate* and can therefore only make Streaming Ranked Pairs choose *indeterminate* for a considered edge, satisfying the induction hypothesis.

Importantly, the weights of these extra edges upper bound their true (unknown) weights. And any cycle that would go through (an) as-yet-unseen transaction(s) maps to one (using *indeterminate* edges) through the future vertex v. As such, the set of paths considered in Streaming Ranked Pairs is always a superset of that considered by Ranked Pairs, and the difference between these sets is always made up of *indeterminate* edges.

As such, when the streaming algorithm does not choose to leave an edge as *indeterminate*, the streaming algorithm makes the same decisions as the non-streaming algorithm. Thus, the induction hypothesis holds for (tx_i, tx_j).

The induction hypothesis clearly holds for the first edge considered, so the lemma holds. □

The same argument shows that Algorithm 2 is monotonic.

Corollary 4.6.9. *Algorithm 2 is monotonic.*

Proof. The proof of Lemma 4.6.8 actually shows that an edge is marked *determinate* or rejected only if Ranked Pairs would make that decision on the edge on *any* counterfactual scenario that extends the input to the algorithm.

□

Lemma 4.6.8 directly implies that the output of Algorithm 2 matches that of Ranked Pairs.

Theorem 4.6.10. *The output of Algorithm 2 matches an initial segment of the ordering output by the non-streaming Ranked Pairs (on the counterfactual of Lemma 4.6.8).*

Proof. Note that every transaction bordering on an indeterminate edge must come strictly after (in the topological ordering) every transaction in the output of the algorithm.

By Lemma 4.6.8, then regardless of whether or not the indeterminate edges are chosen in Ranked Pairs, every transaction bordering an indeterminate edge must come after (in the true output of Ranked Pairs) all of the transactions output by the algorithm.

Within the set of output transactions, again because of Lemma 4.6.8, transactions must be ordered according to the true output of Ranked Pairs. □

Ranked Pairs has the property that, for any set of inputs $(\sigma_1, \ldots, \sigma_n)$, if it is run on a restriction of the inputs to the transactions that appear as an initial segment of its output on $(\sigma_1, \ldots, \sigma_n)$, its output is unmodified on the restricted inputs (for completeness, we give a proof in §B.1). As Algorithm 2 is monotonic, its output exactly matches the Ranked Pairs ordering when the input votes are restricted in this way.

4.7 Liveness and Efficiency

ALGORITHM 3: Streaming Ranked Pairs, Version 2

```
Input: An ordering vote from each replica $\{\sigma_i\}_{i\in[n]}$
Input: $\hat{H}$, the graph computed in the preceeding invocation of Algorithm 3
$\hat{G} = (V, E, w) \leftarrow \hat{G}(\sigma_1, ..., \sigma_n)$ ;          /* Compute streamed ordering graph */
$H \leftarrow (V, \emptyset)$ ;          /* Initialize empty graph on same vertex set */
foreach edge $(tx, \hat{v})$ and $(\hat{v}, tx)$ with $w(e) = 1$ do
    Add $e$ to $H$, and mark it as indeterminate
end
Add all determinate edges of $\hat{H}$ to $H$, marked determinate
foreach $\gamma$ in $(1, \frac{n-1}{n}, \frac{n-2}{n}, \ldots, 0)$ do
    $E_\gamma \leftarrow \{e \in E \mid w(e) = \gamma\}$, ordered arbitrarily
    repeat
        $e = (tx_i, tx_j) \leftarrow$ first element of $E_\gamma$
        $U_{tx_i,tx_j} \leftarrow (V \setminus (R_{tx_i} \cup P_{tx_j})) \cup \{\hat{v}\}$
        If there is a directed path in $H$ restricted to $U_{tx_i,tx_j}$ from $tx_j$ to $tx_i$ where every edge
          is determinate, then do not include the edge in $H$. Remove the edge from $E_\gamma$.
        Else if there is no directed path in $H$ restricted to $U_{tx_i,tx_j}$ from $tx_j$ to $tx_i$ where
          every edge is either determinate or indeterminate, add $(tx_i, tx_j)$ to $H$ and mark it
          as determinate. Remove the edge from $E_\gamma$.
        Otherwise, defer $(tx_i, tx_j)$ to the end of the ordering on $E_\gamma$.
    until $E_\gamma$ is empty or no progress is made in one full loop over $E_\gamma$;
    Mark all remaining edges in $E_\gamma$ as indeterminate
end
Output: The topological sort of $H$ (with sorting comparisons restricted to determinate
        edges), up to but not including the first transaction $tx$ bordering an
        indeterminate edge
```

4.7.1 Efficient Marginal Runtime

In a real system, a ranking algorithm would be run repeatedly. Rather than recompute the streamed ordering graph and the status of every edge in each invocation, Algorithm 2 could use the output of a past invocation, restricted to the decisions that were *determinate*, as an oracle. Corollary 4.6.9 implies that *determinate* choices are consistent with any extension of the input, so this oracle does not change the output. However, the runtime now depends only on the number of new edges in the streamed ordering graph (§B.6).

4.7.2 Protocol Liveness

Algorithm 2 is not asymptotically live. §B.5 gives an instance on which the algorithm would never output any transaction.

That construction uses an adversarial tiebreaking rule, but Ranked Pairs requires only some tiebreaking rule. Our algorithm can therefore generate a tiebreaking rule dynamically. Specifically, when the algorithm visits edges of weight $\gamma = \frac{k}{n}$, it defers edges that would become indeterminate to the end of the ordering among edges of that weight.

Algorithm 3 constructs a (partial) ordering between edges that breaks ties between edges of the same weight. The output of this algorithm is, by the same arguments as in §4.6.2, consistent with the output of non-streaming Ranked Pairs, when using an tiebreaking rule consistent with this edge ordering. The oracle mechanism of §4.7.1 passes this tiebreaking information from one run of Algorithm 3 to the next.

Lemma 4.7.1. *Whenever Algorithm 3 includes a determinate edge in H (resp. excludes an edge from H), the non-streaming Ranked Pairs, when using the implied ordering, would make the same decision.*

Proof. The argument proceeds as in Lemma 4.6.8. If Algorithm 2 were given the tiebreaking rule implied by the ordering induced by Algorithm 3, it would visit edges in the same ordering as this version of the streaming algorithm. As such, its output would be consistent with the non-streaming Ranked Pairs using this implied ordering (by Theorem 4.6.10). □

Crucially, this small change to the ordering enables the following structural lemma. We start with a useful definition.

Definition 4.7.2. *An edge $(\overline{tx}, \overline{tx'})$ is contemporary to an edge (tx, tx') if and only if $\overline{tx}$ and $\overline{tx'}$ are contained in $U_{tx,tx'}$.*

Lemma 4.7.3. *If Algorithm 3 marks an edge (tx_i, tx_j) with weight $\frac{k}{n}$ as indeterminate, there must be a path from tx_j to tx_i contained in U_{tx_i,tx_j} that contains an indeterminate edge of weight at least $\frac{k+1}{n}$.*

By contrast, the best bound in Algorithm 2 is that there exists a contemporary indeterminate edge of weight $\frac{k}{n}$.

Proof. By construction of the algorithm, if an edge (tx_i, tx_j) is marked as *indeterminate*, then there must be a path in U_{tx_i,tx_j} already chosen (when this edge is visited) of *determinate* and *indeterminate* edges of weight at least $\frac{k}{n}$. Furthermore, any *indeterminate* edge must be of weight strictly more than $\frac{k}{n}$. When the algorithm looks for such a path, it has not yet marked any edges of weight $\frac{k}{n}$ as *indeterminate*, and if no such path exists, then the edge would not be marked as *indeterminate*. □

Lemma 4.7.3 implies that for any indeterminate edge, there is sequence of contemporary indeterminate edges of strictly increasing weights that causes the edge to be indeterminate.

Lemma 4.7.4. *Suppose Algorithm 3 marks an edge e_1 of weight γ as indeterminate. Then there exists a sequence of indeterminate edges $e_1, \ldots e_k$ of strictly increasing weight such that for each pair of edges $e_i = (tx_i, tx_j)$ and $e_{i+1} = (tx'_i, tx'_j)$ with e_{i+1} contemporary to e_i, tx'_i and tx'_j are present in U_{tx_i,tx_j}, and the source of e_k is $\hat{v}$.*

Proof. Follows from repeated application of Lemma 4.7.3.

The algorithm starts with only edges leaving the "future" vertex $\hat{v}$ marked as *indeterminate*. These initial edges are the only *indeterminate* edges with weight 1, and thus are the only ones that can form the root of any chain. □

None of these chains can be longer than n edges, as each step increases the weight by at least $\frac{1}{n}$. This bound implies that Algorithm 3 eventually outputs every transaction.

Theorem 4.7.5. *Algorithm 3 is asymptotically live.*

Proof. Let $(\hat{\sigma_1}, \ldots, \hat{\sigma_n})$ be any set of (countably infinite) ordering preferences, and let tx be any transaction in those orderings.

Consider the set of all chains of edges $(e_1, \ldots, e_k)$ where e_{i+1} is contemporary to e_i (and of higher weight) and e_1 is adjacent to tx, and $k \leq n$. Because each $U_{tx,tx'}$ is finite, the number of such chains must be finite, and the number of transactions that appear in any of these chains must be finite.

On any input to Algorithm 3 that is a finite truncation of $(\hat{\sigma_1}, \ldots, \hat{\sigma_n})$, if an edge adjacent to tx is left *indeterminate*, then Lemma 4.7.4 implies that there must be a chain of *indeterminate* edges on this input with the first adjacent to tx, the last adjacent to $\hat{v}$, and each edge contemporary to the previous. This chain, with the last edge dropped, must be one of the chains considered above. As such, there are only a finite number of transactions that could be in the last edge adjacent to $\hat{v}$. For each such transaction tx', there can only be an edge from $\hat{v}$ to tx' in the streamed ordering graph if there is some other transaction in the input that is ahead of tx' in at least one replica's vote. There can only be a finite number of these transactions.

Taking a union over a finite number of finite sets gives a finite set of transactions. There must therefore be some bound M_{tx} such that all of these transactions appear in every replica's vote if $(\hat{\sigma_1}, \ldots, \hat{\sigma_n})$ is truncated to (at least) the first M_{tx} elements.

A transaction tx is not output by Algorithm 3 if the topological sort in the last step puts a transaction tx' adjacent to an *indeterminate* edge ahead of tx. But there are only a finite number of transactions that might ever be ahead of tx (specifically, P_{tx} and Q_{tx}).

Let $N_{tx} = \max_{tx' \in P_{tx} \cup Q_{tx}} M_{tx'}$. Then if $(\hat{\sigma_1}, \dots, \hat{\sigma_n})$ is truncated to (at least) the first N_{tx}, then tx must appear in the output of Algorithm 3. □

Additionally, under network synchrony, Lemma 4.7.4 bounds the temporal delay between when a transaction is sent by a client and when Algorithm 3 adds it to its output.

Assumption 4.7.6 (Synchronous Network)**.** *If a client sends a transaction tx at time t, all replicas include tx in their ordering votes before time $t + \Delta$.*

As such, replicas can disagree on the ordering between tx and tx' only if they were sent at roughly the same time.

Observation 4.7.7. *Consider any two transactions tx_i and tx_j sent at times t_i and t_j, respectively. If $t_i + \Delta < t_j$, then the weight of the edge (tx_i, tx_j) is 1.*

Proof. If $t_i + \Delta < t_j$, then every replica receives and commits to a vote on tx_i before tx_j is sent, so tx_j must come after tx_i in every replica's vote. □

Lemma 4.7.8. *Consider any three transactions tx, tx_i, and tx_j received by all replicas, that were sent by clients at times t, t_i, and t_j, respectively. Suppose $tx \in U_{tx_i, tx_j}$, and that the weight of (tx_i, tx_j) is not 0.*

Then $t_j - \Delta \leq t \leq t_j + 2\Delta$.

Proof. By construction, $tx \notin R_{tx_i}$, so $t \leq t_i + \Delta$. Furthermore, $tx \notin P_{tx_j}$, so $t + \Delta \geq t_j$. Furthermore, by Observation 4.7.7, $t_j + \Delta \geq t_i$.

Thus, $t_j - \Delta \leq t \leq t_i + \Delta \leq t_j + 2\Delta$. □

This lemma and Lemma 4.7.4 give an overall time bound.

Theorem 4.7.9. *A transaction tx is contained in the output of the algorithm of Algorithm 3 after at most $(n+1)\Delta$ time.*

Proof. Let the current time be T, and let tx be sent at time t. Without loss of generality, assume tx has been voted on by every replica.

If an *indeterminate* edge exists to tx, then there exists some transaction tx' (sent at t') that has not been voted on by every replica, but which preceeds tx in some replicas' votes. As such, $T - \Delta \leq t'$. Then, as tx does not fully preceed tx', it must be the case that $t' - \Delta \leq t$ (so $T - 2\Delta \leq t$).

Lemma 4.7.4 implies a sequence of *indeterminate* edges of strictly increasing weight from this edge to one adjacent to $\hat{v}$.

Consider two adjacent edges $(tx_i, tx_j), (tx'_i, tx'_j)$ in this chain, send at times t_i, t_j, t'_i, t'_j, respectively. By Lemma 4.7.8, it must be the case that $t_j - \Delta \leq t'_j$. Adding this bound over each link in the chain shows that $T - 2\Delta - (k-1)\Delta - \Delta \leq t$ (where the last Δ comes from bounding the delay of the transaction in the last edge adjacent to $\hat{v}$).

A sequence can be of length at most $n-1$.

As such, after time $(n+1)\Delta$, every edge adjacent to tx is either rejected or included and marked *determinate*.

To ensure that tx is included in the output of Algorithm 3, it suffices to ensure that every transaction adjacent to an *indeterminate* edge must come after tx. This is guaranteed by waiting for an additional Δ time. □

4.7.3 Trading Accuracy for Liveness

The $O(n\Delta)$ bound in Theorem 4.7.9 comes from the fact that a chain of indeterminate edges can have $O(n)$ length, which, in turn, follows from the fact that weights have a granularity of $\frac{1}{n}$. Reducing this granularity by rounding weights reduces the maximum length of a chain of indeterminate edges.

Lemma 4.7.10. *If edge weights are rounded to the nearest $\frac{1}{k}$ in the streamed ordering graph (before Algorithm 3 is applied) then a transaction tx is contained in the output after at most $(k+2)\Delta$ time.*

Proof. The argument proceeds exactly as in that of Theorem 4.7.9, except that the length of the chain of *indeterminate* edges is at most k. □

However, rounding weakens fairness guarantees.

Lemma 4.7.11. *Rounding edge weights to the nearest $\frac{1}{k}$ before applying Algorithm 3 achieves $(\gamma, \frac{f}{n} + \frac{1}{2k})$-minimal-batch-order-fairness for all γ and f faulty replicas.*

Proof. The argument proceeds as in Theorem 4.5.1. However, due to the rounding and the faulty replicas, the weight on an edge observed by the algorithm might be up to $\frac{f}{n} + \frac{1}{2k}$ different from the true observed weight (as opposed to $\frac{f}{n}$, as in Theorem 4.5.1). □

4.8 Protocol Instantiation

The algorithms thus far have assumed that every replica will eventually vote on every transaction. Any faulty replica, however, could exclude a transaction from its output, leaving the transaction (and the protocol) permanently stuck.

The main idea to resolve this difficulty is that if a replica refuses to vote on a transaction, then that replica must be faulty. If a replica is faulty, then it suffices for the rest of the replicas to make up a vote on its behalf.

We give here an example method for instantiating our sequencing protocol on top of existing protocol components, although this is far from the only method.

As a base layer, we require some consensus protocol, such as [316, 233, 114, 245], that proceeds in *rounds*. At any time, every replica can submit a continuation of its previous ordering; that is, an ordered list of transactions that extends its ordering vote. The consensus protocol then comes to consensus on a set of received ordering votes. The Streaming Ranked Vote algorithm is run (by every replica) on the set of received votes after every new set of votes is finalized by the consensus protocol. Assuming correctness of the consensus protocol, every replica will run the algorithm on the same input, and thereby produce the same output ordering.

We assume that every replica has a registered public key. Every vote continuation must be cryptographically signed by the associated secret key. Furthermore, every continuation must be accompanied by the hash of the previously submitted continuation, so that if one continuation vote from a replica is lost (perhaps by malicious activity of consensus participants) the replica's ordering cannot be manipulated.

We make a network synchrony assumption, akin to Assumption 4.7.6. If a transaction appears first in some replica's vote in round T of the consensus protocol, then we assume that there is some known bound k such that every other honest replica will have time to receive and submit a vote on the transaction before the end of round $T + k$. As such, after every round r, the protocol computes the set of all transactions that have not received votes from every replica but were initially added at or before round $r - k$. For each of these transactions tx, the protocol appends a vote for tx at the end of a replica's current vote sequence if tx is not already present When appending multiple transactions to a replica's vote, transactions can be sorted deterministically (i.e. by hash).

This protocol therefore guarantees that every transaction will receive a vote from all replicas within at most k rounds after it is first submitted to the protocol. In this setting, k must be configured in advance by the protocol. k must be set so that every honest replica can be guaranteed to be able to submit a vote on a particular transaction within k rounds. Naturally, the choice of k depends on a (known) bound on network delay and the censorship-resistance (or "chain-quality" [175]) of the consensus protocol.

As an example alternative, one could also use the consensus protocol, based on atomic broadcast and a "Set Byzantine Agreement" primitive used in Aequitas [209]. This protocol waits until all replicas, if behaving honestly, are able to add a transaction tx to their ordering votes, and then invokes an agreement protocol to come to consensus on the set of replicas U_{tx} which have the transaction in their ordering votes. This set may not be the set of all replicas. To use our ordering protocol on this consensus protocol, it suffices to make up a vote for the (presumed faulty) replicas not in $U_{tx} \cap U_{tx'}$. Choosing arbitrarily, replicas in $U_{tx} \setminus U_{tx'}$ can be presumed to vote $tx \preccurlyeq tx'$ and vice versa, and replicas not in $U_{tx} \cup U_{tx'}$ can be presumed to vote by comparing deterministic hashes of tx and tx'.

4.9 Conclusion

We study the problem of ordering transactions in a distributed system, which we argue forms a streaming analogue of the classic social choice problem. Careful analysis of how incomplete information propagates through the Ranked Pairs algorithm allows us to instantiate Ranked Pairs in this streaming setting. Deliberate manipulation of the algorithm's tiebreaking rule guarantees that the algorithm always outputs every transaction. Under network synchrony, the algorithm outputs every transaction after a bounded amount of time.

We then extend the notion of γ-batch-order-fairness of prior work to require that output batches are minimal in size (so the definition is not vacuous). This must be done carefully in the presence of faulty replicas, and leads to our definition of (γ, δ)-minimal-batch-order-fairness.

Finally, we show that Ranked Pairs voting satisfies $(\gamma, \frac{f}{n})$-minimal-batch-order-fairness for every γ simultaneously and for any number of faulty replicas f. For comparison, prior work must fix a choice of γ and a bound on the number of faulty replicas in advance, and can only satisfy γ-batch-order-fairness for that γ. Fairness guarantees smoothly weaken as the number of faulty replicas increases.

Chapter 5

Finding the Right Curve: Optimal Design of Constant Function Market Makers

5.1 Introduction

Agents in any economic system need to be able to exchange one asset for another efficiently. Some assets are frequently traded by many market participants, and for these assets, a seller offering a reasonable price can likely find a buyer quickly and vice versa. However, not every pair of assets is traded frequently, and sellers in these markets might have to wait a long time to find a buyer or accept a highly unfavorable price. The role of a *market-maker* is to fill this gap — to facilitate easy and rapid trading between pairs of assets for which otherwise there is very little trading activity. Market-makers trade in both directions on the market, buying and selling assets when traders arrive at the market [62]. In this sense, market-makers facilitate asynchronous trading between buyers and sellers, thereby increasing the market *liquidity* between two assets.

Our topic of study is a subclass of automated market-making strategies known as *Constant Function Market Makers* (CFMMs). A CFMM maintains reserves of two assets X and Y, provided by a so-called *liquidity provider (LP),* and makes trades according to a predefined *trading function* $f(x, y)$ of its asset reserves (the eponymous "constant function"); specifically, a CFMM accepts a trade $(\Delta x, \Delta y)$ from reserves (x, y) to $(x - \Delta x, y + \Delta y)$ if and only if $f(x - \Delta x, y + \Delta y) = f(x, y)$. CFMMs earn revenue by charging a small commission on each trade (i.e. creating a bid-ask spread) but are subject to several associated expenses [63], such as the costs of maintaining the asset inventory and adverse selection by arbitrageurs (i.e., stale quote sniping). The loss of the LP relative to the counterfactual strategy of “buy-and-hold” is referred to as the “divergence loss” [244].

Automated market-making has long been an important topic of study [72, 180, 258], but CFMMs have recently become some of the most widely used exchanges [55, 56, 241, 157] within the modern Decentralized Finance (DeFi) ecosystem [305].

Their rise in current blockchain systems is in large part due to their computational simplicity. In existing systems that execute transaction sequentially, the amount of computation allowed for any one transaction is inherently limited. The first of these systems that attained widespread usage [55], for example, used a trading curve of $f(x, y) = xy$. Executing a transaction on this CFMM requires no more computation than just evaluating $f(\cdot, \cdot)$ twice (two multiplications), adding Δ_X and Δ_Y, and checking that the value of f is the same before and after the addition. [1]

CFMMs have also been widely deployed in prediction markets for aggregating opinions [191, 122]. For completeness, §C.1 gives the precise relationship between prediction markets and CFMMs.

Example 5.1.1 (Real-World CFMMs)**.**

1. *The decentralized exchange Uniswap [55] uses the product function $f(x, y) = xy$.*
2. *The Logarithmic Market Scoring Rule (LMSR) [191], used extensively to design prediction markets, corresponds to a CFMM with trading function $f(x, y) = (1 - e^{-x}) + (1 - e^{-y})$ [273].*
3. *The trading function $f(x, y) = xe^y$ has powered automated storefronts [28] within Minecraft [248].*

These systems have facilitated billions of USD worth of trade volume [45] per day. In spite of this, a complete, formal understanding of the tradeoffs inherent in CFMM design is missing in the literature.

Our goal, therefore, is to explain what guides a CFMM designer to choose one trading function over another. We provide an optimization framework which compiles a market-maker's beliefs on future prices into an optimal CFMM trading function, making substantial progress on an important open problem [276].

5.1.1 Our Contributions

We develop a convex optimization framework that translates an LP's beliefs about future asset valuations into an optimal choice of CFMM trading function. We show that a unique trading function always maximizes an LP's expectation of the fraction of trades that a CFMM can settle

[1] The actual implementation [32] checks that the value of $f(\cdot, \cdot)$ does not decrease, to handle approximation due to rounding. Rational traders leave $f(\cdot, \cdot)$ increased as little as possible.

The contracts also charge transaction fees, and do some additional recordkeeping, which we ignore here. These additional features are likewise designed to use as little computational work as possible.

A more recent implementation [56] simulates more complicated curves (as in §5.5.2), by using the constant product curve defined piecewise on a finite ranges of prices. The granularity of this discretization can affect the computational cost of operating the CFMM, as discussed in [165]. While our model can be naturally modified to optimize over the subclass of curves implementable with a given discretization, we do not explicitly consider the cost of the complexity of evaluating $f(\cdot, \cdot)$ in our model.

(§5.3.2). Furthermore, our framework is versatile such that it can model a wide variety of real-world concerns, including fee revenue (§5.6), divergence loss (§5.6.2), so-called "Loss-Versus-Rebalancing" [244] (§5.6.2), and models of price dynamics (§5.3.4). To the best of our knowledge, this is the first unified framework for analysing and optimizing CFMMs for various objectives for a given belief function and therefore provides a guide to prospective LPs interested in using a CFMM.

We model an LP's beliefs as a joint distribution on the future prices of the two assets with regard to a numeraire (§5.5). This belief could, for example, be generated from a price dynamics model. As one might expect, the optimization problem concerned only with maximizing the fraction of trades settled depends only on the distribution of the *ratio* of prices. However, expressing beliefs on future prices as a joint distribution enables optimizing for profit and loss through the same framework.

We measure the liquidity of the CFMM trading function as the amount of capital implicitly allocated for market-making at a given spot exchange rate. Specifically, the liquidity of a CFMM is the ratio between the size of a trade and the percentage change in the spot exchange rate (§5.3.1).

We analyze the steady-state dynamics of trade requests on a CFMM and arrive at a notion of "CFMM inefficiency" that approximates the probability that a CFMM cannot satisfy a trade request. CFMM inefficiency is a function of the inverse of the CFMM's liquidity. Ultimately, we find that many complex objective functions considering an LP's profit and loss are linear combinations of CFMM liquidity and CFMM inefficiency.

Careful analysis of the KKT conditions of our convex program allows us to invert the problem; given an arbitrary CFMM trading function, we can construct an explicit equivalence class of beliefs for which the given function is optimal. The main technique involves analysis of the KKT conditions of an optimization problem over an infinite-dimensional Banach space. We obtain closed-form solutions to the optimal CFMM designs for several important belief functions and objective functions. When not closed-form, the solution is still computationally tractable.

Our framework helps explain the choice of CFMM trading functions deployed in practice. In many cases, the optimal CFMM revealed by our framework matches the informal intuitions of practitioners. For example, the Uniswap V2 [55] protocol in DeFi was designed using the constant product $f(xy) = xy$ CFMM with the motivation that the available liquidity must be spread evenly across all exchange rates. In our framework, $f(xy) = xy$ is the optimal CFMM trading function for the uniform belief function with the objective of minimising the expected CFMM inefficiency.

Figure 5.1 shows the trading functions and Figures 5.6 represent the belief functions for which the constant product, LMSR, constant weighted-product, and Black-Scholes-based CFMMs, respectively, are optimal for minimizing the CFMM inefficiency.[2]

§5.2 formally defines a CFMM and gives some basic properties. §5.3 studies the steady-state dynamics of a CFMM and defines CFMM liquidity and inefficiency. §5.4 gives our convex optimization framework and analyzes its KKT conditions. §5.5 studies the beliefs implicit behind real-world

[2]We provide, at the GitHub repositiry `https://github.com/gramseyer/cfmm-liquidity-optimization`, a Python script to generate the optimal CFMM trading functions for any user-defined belief function.

CFMM	Trading Function	Liquidity $L(p)$	Belief $\psi(p_X, p_Y)$
Constant product	xy	$\frac{1}{2}\sqrt{p}$	1
Constant weighted product	$x^\alpha y$	$\frac{\alpha}{\alpha+1}p^{\frac{\alpha}{\alpha+1}}$	$\left(\frac{p_X}{p_Y}\right)^{\frac{\alpha-1}{\alpha+1}}$
LMSR based	$1-e^{-x}+1-e^{-y}$	$\frac{p}{1+p}$	$\frac{p_X p_Y}{(p_X+p_Y)^2}$
Lognormal prior based	As in Figure 5.1	$\sqrt{e^{\frac{-(\ln p)^2}{2\sigma^2}}}$	$\frac{p_X}{p_Y} \sim lognormal(0, \sigma^2)$
Black-Scholes based	As in Figure 5.1	Not closed form	As in equation 5.1.

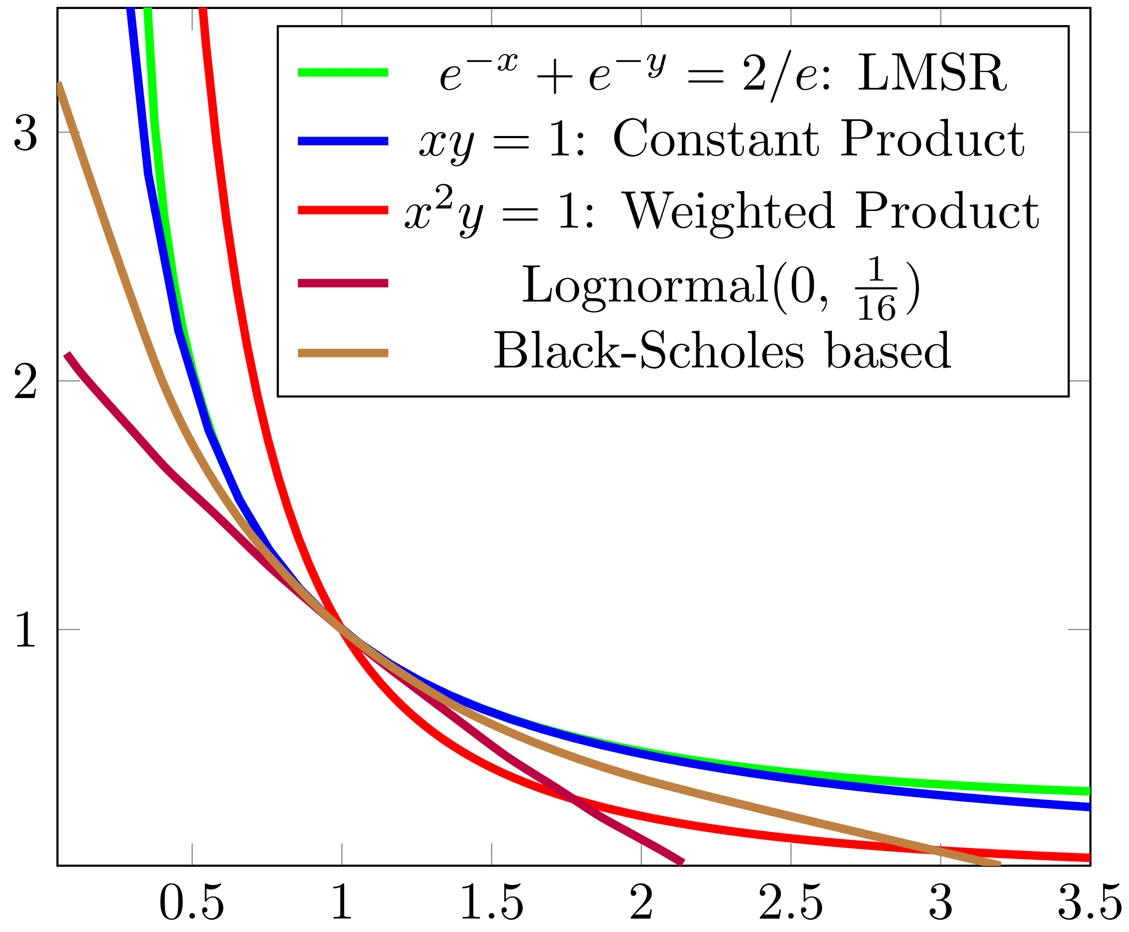


Figure 5.1: Some natural or widely used CFMM trading functions. The lognormal belief function arises when we consider a snapshot of the Black-Scholes process at a future time. The entire Black-Scholes process can be considered (in expectation) for the purpose of our optimization framework via time-discounting (in the plot, the discounting parameter is 1). It can then be compiled into a single belief function as described in §5.3.4. Our Python script in the Github repository https://github.com/gramseyer/cfmm-liquidity-optimization computes the belief function for the Black Scholes model. For any user-submitted belief function, our script also finds the optimal CFMM trading functions for minimizing CFMM inefficiency.

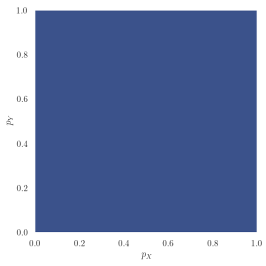


Figure 5.2: Constant belief function – leads to the constant product market maker.

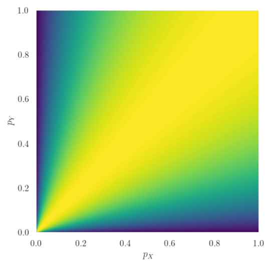


Figure 5.3: Belief function $\psi(p_X, p_Y) = \frac{p_X p_Y}{(p_X+p_Y)^2}$ – leads to the LMSR based market maker

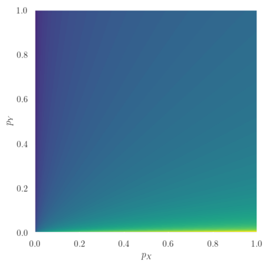


Figure 5.4: Belief function $\psi(p_X, p_Y) = \left(\frac{p_X}{p_Y}\right)^{1/5}$ – leads to the weighted product market maker.

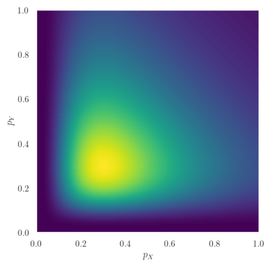


Figure 5.5: Lognormal belief function – leads to the Black-Scholes based CFMM after time-discounted aggregation of the belief function.

Figure 5.6: Belief functions on future prices of the underlying assets relative to a numeraire. The plots are on beliefs defined on the range $(0, 1]$ – this is without loss of generality per Corollary 5.4.6.

CFMM deployments. Finally, §5.6 shows how to add consideration for profit and loss to our framework, and qualitatively studies how these considerations change the optimal trading function.

5.1.2 Related Work

The closest line of work [165, 166, 112, 193, 86] to our results is that which considers profit-maximizing market-making strategies which can be implemented via the Uniswap v3 [56] protocol. Additionally, [166, 112, 86] design "rebalancing" strategies for the LPs, wherein they effectively modify the CFMM trading function periodically. By contrast, we consider designing CFMM trading functions from first principles and do not rely on the Uniswap v3 framework. We also do not consider rebalancing the CFMM trading function in this work. A non-exhaustive list of papers in this line is:

- Fan et al. [165], study the question of maximizing risk-adjusted profit for LPs while accounting for the gas fee for traders. Their model assumes that all trading on a CFMM occurs only in response to price movements on an external market (i.e. arbitrageurs realigning the CFMM spot price to the external market). Their model suggests that risk-neutral LPs must allocate all of their capital at a single price point (§4.2, [165]), while ours better explains the choices of practitioners.

- Zhou et al. [166] study dynamic liquidity allocation strategies for risk-adjusted fee revenue maximization, but do not consider the "divergence loss" incurred in the process.

- Cartea et al. [112] decompose the CFMM divergence loss into two components – the convexity cost (loss due to arbitrage) and the opportunity cost (the cost of locking up capital). They give a stochastic optimal control-based closed-form strategy for a profit-maximizing LP.

- Heimbach et al. [193] model liquidity positions on Uniswap V3 and perform a data-based analysis of the risks and returns of LPs as a function of the volatility of the underlying assets.

Similar to [112], Milionis et al. [244] show that a part of the divergence loss corresponds to the market risk and can be hedged by a rebalancing strategy; the remainder of the divergence loss corresponds to the profit made by arbitragers trading against the CFMM – they call this loss the LVR (loss-vs-rebalancing). When the variance of the price of X relative to Y is σ^2, they show that the rate of accrual of LVR (what they call the instantaneous-LVR) is $\sigma^2 p^2 |x'(p)|$, where $x'(p)$ denotes the rate of change of x in the CFMM with respect to the price p under perfect arbitrage. Since the LVR is a linear function of our notion of liquidity, our convex optimization framework can accommodate the LVR as a cost for the LP in the objective function.

Automated market-making has also been studied extensively in the context of prediction markets [191, 122, 121]. The theory of CFMMs and the dynamics around trading with CFMMs have been studied in DeFi [65, 69, 67, 111, 88, 95], and many different DeFi applications have been deployed or proposed using different CFMM trading functions [55, 56, 241, 66].

5.2 Preliminaries

Definition 5.2.1 (CFMM)**.** *A CFMM trades between two assets X and Y, and has a set of asset reserves — x units of X and y units of Y. Its trading rule is defined by its trading function $f(\cdot,\cdot)$ such that it accepts a trade of Δ_X units of X in exchange for Δ_Y units of Y if and only if $f(x,y) \leq f(x-\Delta_X, y+\Delta_Y)$.*

Rational traders interacting with the CFMM choose Δ_X and Δ_Y such that the inequality of Definition 5.2.1 is tight. [3]

All of the CFMM trading functions discussed in this thesis have the following properties.

Assumption 5.2.2. *A trading function $f(\cdot,\cdot) : \mathbb{R}^2_+ \to \mathbb{R}$ is continuous, non-negative, increasing in both coordinates, and strictly quasi-concave. Further, it is defined only on the non-negative orthant.*

The assumption that f is increasing, quasi-concave, and never holds a short position in any asset (and is therefore only defined on the non-negative orthant) is standard in the literature, and required for a CFMM to not be *obviously* exploitable [65, 281].

We assume strict quasi-concavity for clarity of exposition. The CFMM's trading function implicitly defines a marginal exchange rate (the "spot exchange rate") for a trade of infinitesimal size.

Definition 5.2.3 (Spot Exchange Rate)**.** *At asset reserves (x_0, y_0), the spot exchange rate of a CFMM with trading function f is $-\frac{\partial f}{\partial X} / \frac{\partial f}{\partial Y}$ at (x_0, y_0).*

When f is not differentiable, the spot exchange rate is any subgradient of f. When $x_0 = 0$, the spot exchange rate is $[-\frac{\partial f}{\partial X} / \frac{\partial f}{\partial Y}, \infty)$, and when $y_0 = 0$, the spot exchange rate is $[0, -\frac{\partial f}{\partial X} / \frac{\partial f}{\partial Y}]$.

These definitions directly lead to some useful observations. We give the proofs in Appendix C.3.1.

Observation 5.2.4. *If f is strictly quasi-concave, then for any constant $K > 0$ and spot exchange rate p, there is a unique point (x, y) where $f(x, y) = K$ and p is a spot exchange rate at (x, y).*

Observation 5.2.5. *Under Assumption 5.2.2, for a given constant function value K, the amount of Y in the CFMM reserves uniquely specifies the amount of X in the reserves, and vice versa.*

Observations 5.2.4 and 5.2.5 imply that the amounts of X and Y in the CFMM reserves can be written as functions $\mathcal{X}(p)$ and $\mathcal{Y}(p)$ of its spot exchange rate for the trading function equals constant K.

In the rest of the discussion, we describe CFMM reserve states by the amount of Y in the reserves.

Observation 5.2.6. *$\mathcal{Y}(p)$ is monotone nondecreasing.*

[3] Anything else is giving up free money. But this is the precise condition checked by existing CFMM implementations [33]. Slight increases in the value of the trading function are inevitable, due to rounding errors when asset amounts are represented as integers (as is done in nearly all public blockchains [135]).

5.3 Model

As used in Definition 5.2.3, we adopt the notation wherein exchange rates are given as the rate of a unit of X in terms of Y (i.e., a trade of x units of X for y units of Y implies an exchange rate of $p' = \frac{y}{x}$). Unless specified otherwise, p refers to the CFMM spot exchange rate, $\hat{p}$ denotes the exchange rate in an external market, and p' denotes the exchange rate of a particular trade.

We now turn to our trading model and our formulation of market liquidity.

Definition 5.3.1 (System Model)**.**

1. *There are two assets X and Y, and a relatively liquid "primary" external market that provides a (public) reference exchange rate $\hat{p}$ between X and Y.*
2. *An LP creates a CFMM that trades between X and Y by providing an initial set of reserves and choosing a CFMM trading function.*
3. *Whenever the reference exchange rate $\hat{p}$ on the external market changes, arbitrageurs immediately realign the CFMM's spot exchange rate p with the reference exchange rate.* [4]
4. *At each time step, a trade request arrives with probability $q : 0 < q < 1$ (Definitions 5.3.3 and 5.3.5).*

We assume, however, that for small fluctuations in the CFMM spot exchange rate resulting from small trades, arbitrageurs do not realign the CFMM spot exchange rate. This assumption is reasonable since the trading fee and other associated costs (e.g., gas fee in DeFi) make such an action unprofitable.

Since the reference exchange rate is public knowledge, traders using the CFMM can compare the exchange rate that a CFMM offers with the reference rate. This difference is the *slippage* of a trade.

Definition 5.3.2 (Slippage)**.** *The exchange rate of a trade of y units of Y for x units of X is $p' = y/x$. Relative to a reference exchange rate of $\hat{p}$ units of Y per X, the* slippage *of this trade is $(p' - \hat{p})/\hat{p}$.*

Traders in our model are willing to tolerate a fixed amount of maximum slippage ε.

Definition 5.3.3 (Trade Request)**.** *A* Trade Request *with a CFMM is a request to SELL or BUY k units of X or Y, on the condition that the slippage of the trade is at most ε relative to the reference exchange rate $\hat{p}$ — in other words, a trade request is a tuple (SELL or BUY, X or Y, k, $\hat{p}$, ε).*

Definition 5.3.4 (Trade Success)**.** *A trade request buying X for Y with maximum slippage ε succeeds if and only if the CFMM can satisfy the entire trade with an exchange rate p' and, for the reference*

[4]There is always a strictly profitable arbitrage trade to be made when the CFMM's spot exchange rate differs from the reference exchange rate [68]; this phenomena is akin to "stale quote sniping" in traditional exchanges [82].

exchange rate $\hat{p}$, $p'/\hat{p} \leq 1+\varepsilon$. Similarly, a trade request selling X for Y succeeds if and only if the CFMM can satisfy the entire trade with an exchange rate p' such that $p'/\hat{p} \geq 1/(1+\varepsilon)$.

Trade requests are not partially fulfilled. Failed requests are not retried and are deleted. If a request succeeds, the CFMM transfers assets accordingly. Otherwise, the CFMM's reserves are unchanged. This notion of trade success mirrors the operation of CFMMs in practice; users supply a trade size, exchange rate, and slippage parameter when submitting a trade request (e.g. [25, 1]).

Putting these definitions together gives the trading model of our study.

Definition 5.3.5 (Trade Model)**.** *There exists a static (in the short term) reference exchange rate $\hat{p}$. The size of the trade request is drawn from distribution size$(\cdot)$. The choice of X or Y is arbitrary, but the trade is for BUY or SELL with equal probability. Each request has the same maximum slippage ε.*

Every successful trade changes the reserves of the CFMM – this model induces a Markov chain on the state of the CFMM's reserves. We assume that the Markov chain, at a given reference exchange rate, has sufficient time to mix before the reference exchange rate changes. Natural restrictions on the distribution of the trades (made explicit below) make this Markov chain ergodic. We study, therefore, the expected fraction of trade requests that a CFMM can satisfy when its state is drawn from the stationary distribution of this Markov chain (we formalize this notion in Definition 5.3.15).

5.3.1 Liquidity

Informally, a CFMM with high *liquidity* at a given exchange rate can sell many units of X before its spot exchange rate changes substantially. Definition 5.3.6 captures precisely the set of asset reserve states of a CFMM in which the CFMM's spot exchange rate p is at most a $1+\varepsilon$ factor away from the reference exchange rate $\hat{p}$. Recall from Observation 5.2.4 that the amount of asset Y in a CFMM's reserves can be expressed as a function $Y(p)$ of the CFMM spot exchange rate p.

Definition 5.3.6 ($L_\varepsilon(\hat{p})$)**.** *$L_\varepsilon(\hat{p})$ is the interval $\left[\mathcal{Y}(\frac{\hat{p}}{(1+\varepsilon)}), \mathcal{Y}(\hat{p}(1+\varepsilon))\right]$.*

By Observation 5.2.6, $\mathcal{Y}(\frac{\hat{p}}{(1+\varepsilon)}) \leq \mathcal{Y}(\hat{p}(1+\varepsilon))$, so $L_\varepsilon(\hat{p})$ is always well-defined.

Recall that $\hat{p}$ is the exchange rate of a unit of X in terms of Y. As motivation for the choice of this definition, consider the case where X is a volatile asset and Y is the base numeraire currency. Here, Definition 5.3.6 precisely captures the amount of *capital* allocated to market-making in a range where the spot exchange rate of the volatile asset X is within a $1+\varepsilon$ factor of its reference exchange rate. In the general case where neither X nor Y is the base numeraire currency, the actual amount of capital (in terms of base numeraire) allocated to market-making at a certain price point $\hat{p}$ depends on the exchange rates of both X and Y in the base numeraire. However, a similar intuition holds.

Since ε is expected to be small in practice, and to facilitate easier analysis in the rest of the paper, it is useful to extend Definition 5.3.6 to study the liquidity at a single exchange rate.

Definition 5.3.7 (Liquidity)**.** *The liquidity at an exchange rate $\hat{p}$, $L(\hat{p})$, is $\lim_{\theta\to 0}\frac{|L_\theta(\hat{p})|}{2\ln(1+\theta)}$.*

Observe that $L(\hat{p})$ naturally captures an allocation of capital to market-making on the full range of exchange rates. Recall from the System Model 5.3.1 (point 3) that the arbitrageurs always realign the CFMM's spot exchange rate to the reference exchange rate. Therefore, here on, we denote the liquidity of a CFMM as a function of its spot exchange rate. Lemma 5.3.8 enables a natural restatement of $L(p)$ in terms of $\mathcal{Y}(p)$ in Lemma 5.3.9. We include the proof of Lemma 5.3.8 in Appendix §C.3.1. Lemma 5.3.9 follows from Definitions 5.3.6 and 5.3.7.

Lemma 5.3.8. *The function $\mathcal{Y}(\cdot)$ is differentiable when the trading function f is twice-differentiable on the nonnegative orthant, f is 0 when $x=0$ or $y=0$, and Assumption 5.2.2 holds.*

Lemma 5.3.9. *If the function $\mathcal{Y}(\cdot)$ is differentiable, then $L(p)=\frac{d\mathcal{Y}(p)}{d\ln(p)}$.*

The definition of liquidity implied by Lemma 5.3.9 is closely related to other definitions of liquidity in the literature. The Uniswap V3 whitepaper [56] uses $\frac{d\mathcal{Y}(p)}{d\sqrt{p}}$, which is equivalent to $L(p)/\sqrt{p}$. Some work that builds strategies for LPs on the Uniswap V3 protocol also adopt the same definition of liquidity [166, 193, 165] as [56]. Milionis et al. [244] use $\frac{-d\mathcal{X}(p)}{dp}$ for liquidity, which is equivalent to $L(p)/p^2$; they also introduce a notion of "instantaneous Loss-Versus-Rebalancing" for the CFMM expected cost of operation. This quantity is proportional to $-\frac{p^2 d\mathcal{X}(p)}{dp}$ (Theorem 1, [244]), which is equivalent to $L(p)$ up to constant multipliers. We conclude this subsection with some convenient facts about $L(p)$.

Observation 5.3.10.

1. *$L(p)=\frac{d\mathcal{Y}(p)}{d\ \ln(p)}=p\frac{d\mathcal{X}(p)}{d\ \ln(1/p)}$ (when $\mathcal{Y}(p)$ is differentiable).*
2. *$|L_\varepsilon(\hat{p})|=\int_{\hat{p}/(1+\varepsilon)}^{\hat{p}(1+\varepsilon)}\frac{L(p)}{p}dp$.*
3. *The amount of Y that enters the CFMM's reserves as the spot exchange rate moves from p_1 to p_2 (for $p_1<p_2$) is $\int_{p_1}^{p_2}\frac{L(p)}{p}dp$.*
4. *The amount of Y in a CFMM's reserves, with current spot exchange rate p_0, is $\mathcal{Y}(p_0)=\int_0^{p_0}\frac{L(p)}{p}dp$.*
5. *The amount of X in a CFMM's reserves, with current spot exchange rate p_0, is $\mathcal{X}(p_0)=\int_{p_0}^{\infty}\frac{L(p)}{p^2}dp$.*

Point 1 follows from the fact that $d\mathcal{X}(p)=pd\mathcal{Y}(p)$ and points 2-5 follow from Lemma 5.3.9.

5.3.2 CFMM Inefficiency

We need an expression approximating the fraction of trade requests a CFMM fails to satisfy. As discussed above, the trading model given in Definition 5.3.5 induces a Markov chain on the state of a

CFMM's reserves. We wish to quantify the expected fraction of trades that fail during the evolution of this chain. We assume that the reference exchange rate changes relatively infrequently (so that this Markov chain has time to mix) and study the chain's stationary distribution.

The precise details of the induced Markov chain (we give an example below and another in Appendix §C.2) depends on the trade size distribution (and instantiation-specific assumptions). However, common to many natural distributions is the phenomenon that (when the reference exchange rate is $\hat{p}$) the chance that a trade request of size k units of Y fails is approximately $\frac{k}{L_\varepsilon(\hat{p})}$. This approximation is closest when the sizes of the trades are much smaller than $L_\varepsilon(\hat{p})$.

Example: Constant Trade Sizes

Definition 5.3.11 (Size-k Trade Distribution)**.** *At each time step, a trade request arrives with probability $q : 0 < q < 1$ and buys or sells k units of Y, where buying or selling is chosen with equal probability (and each request tolerates a constant slippage ε).*

For the rest of this section, assume that the reference exchange rate is some unchanging $\hat{p}$. The requirement that $q < 1$ ensures that the Markov chain is ergodic.

Lemma 5.3.12. *When trades are drawn from the size-k trade distribution (Definition 5.3.11), if the CFMM starts with y_0 units of Y, then the stationary distribution of the induced Markov chain is uniform over the points $\{y_0 + kn \mid n \in \mathbb{Z},\ n_{min} \leq n \leq n_{max}\}$ for some integers $n_{min},\ n_{max}$.*

Furthermore, $n_{max} - n_{min} + 2 \geq \frac{|L_\varepsilon(\hat{p})|}{k} \geq n_{max} - n_{min} - 2$, where $\hat{p}$ is the reference exchange rate.

Proof. The only states reachable from y_0 under the size-k trade distribution are a subset of the points $\{y_0 + kn \mid n \in \mathbb{Z}\}$. A trade of size k and maximum slippage ε fails if the spot exchange rate of the CFMM is already above $\hat{p}(1+\varepsilon)$. Thus, there must be some n_{max} such that $y_0 + kn_{max}$ upper bounds the reachable state space. A similar argument shows that n_{min} must exist. Note that n_{min} and n_{max} always exist, even when $k \gg |L_\varepsilon(p)|$ (in which case $n_{min} = n_{max} = 0$).

By the quasi-concavity of the CFMM trading function, the overall exchange rate of a trade must be between the spot exchange rates before and after the trade. Therefore, for any $n \in \mathbb{Z}$, a trade to sell k units of Y must succeed if $y_0 + kn + k \in L_\varepsilon(p)$. Thus, n_{max} must be such that $y_0 + (n_{max}+1)k \notin L_\varepsilon(\hat{p})$ and $y_0 + (n_{max}-1)k \in L_\varepsilon(p)$. A similar argument holds for n_{min}.

In other words, the set of reachable states is a sequence of discrete points, all but the endpoints of which must be in $L_\varepsilon(\hat{p})$. Thus, $(n_{max}+1) - (n_{min}-1) \geq \frac{|L_\varepsilon(\hat{p})|}{k}$, and $(n_{max}-1) - (n_{min}+1) \leq \frac{|L_\varepsilon(\hat{p})|}{k}$.

The Markov chain, therefore, is a random walk on a finite sequence of points with an equal probability of moving in either direction (remaining in place at the endpoints instead of walking beyond the end). Standard results on Markov chains (e.g. Example 1.12, [224]) show that the Markov chain is ergodic and the stationary distribution is uniform over these points. ☐

Once this Markov chain mixes, therefore, the chance at any timestep that a trade fails is the chance that the trade fails if the CFMM is in a randomly sampled state on this Markov chain.

Lemma 5.3.13. *When trades are drawn from the discrete distribution of size k (Definition 5.3.11), the probability that a trade fails is between* $\min\left(1, \left|\frac{k}{|L_\varepsilon(\hat{p})|-k}\right|\right)$ *and* $\frac{k}{|L_\varepsilon(\hat{p})|+k}$.

Proof. Let $n = n_{max} - n_{min}$, as defined in Lemma 5.3.12. The trade request failure chance is $\frac{1}{n+1}$ (trade requests only fail at the endpoints of the sequence of reachable states).

Since $n+2 \geq \frac{|L_\varepsilon(\hat{p})|}{k}$ and $n-2 \leq \frac{|L_\varepsilon(\hat{p})|}{k}$ (from Lemma 5.3.12), we have that $\frac{k}{|L_\varepsilon(\hat{p})|+k} \leq \frac{1}{n+1} \leq \frac{k}{|L_\varepsilon(\hat{p})|-k}$. □

When $k \ll |L_\varepsilon(\hat{p})|$, the trade failure probability is closely approximated by $\frac{k}{|L_\varepsilon(\hat{p})|}$ where the approximation error is $O\left(\frac{k^2}{|L_\varepsilon(\hat{p})|^2}\right)$. The rest of this work makes the following assumption.

Assumption 5.3.14 (Small Trade Size)**.** *Trade sizes are upper bounded by a constant.*

Assumption 5.3.14 is not required for the model in general – traders can submit trades of size comparable to $|L_\varepsilon(\hat{p})|$ units of Y, which may succeed with non-zero probability. However, the assumption enables an approximation of the trade failure probability.See also that $|L_\varepsilon(\hat{p})|= \int_{\hat{p}/(1+\varepsilon)}^{\hat{p}(1+\varepsilon)} L(p)\, d\ln(p)$ (Observation 5.3.10). If $L(\cdot)$ is relatively constant in a neighborhood of exchange rate $\hat{p}$, then $|L_\varepsilon(\hat{p})| \sim L(\hat{p}) \cdot 2\ln(1+\varepsilon)$, and so, under Assumption 5.3.14, for any ε, the chance that a trade fails is proportional to $k/L(\hat{p})$.

With this in mind, we define the following "CFMM inefficiency" metric.

Definition 5.3.15 (CFMM Inefficiency)**.** *The CFMM's inefficiency at an exchange rate $\hat{p}$, with regard to a trade of size k units of Y, is $\frac{k}{L(\hat{p})}$. The inefficiency of a trade denominated in X is equivalent to the inefficiency of a trade of size $k\hat{p}$ units of Y.*

This metric has important implications for the performance of a CFMM. Consider, for example, a trader submitting a trade request of size k units of Y repeatedly until it succeeds. The expected number of times they have to submit the trade is $1/(1-\frac{k}{L(\hat{p})})$. Apart from being an important metric in itself, the CFMM inefficiency is also a crucial factor when considering the LP's profits, as we will see in §5.6. Since the CFMM inefficiency is a convex function of each $L(\hat{p})$, it can directly be incorporated in the objective function of our optimization framework in §5.4.

5.3.3 A Liquidity Provider's Beliefs

We represent an LP's beliefs on future asset prices as a function of a base "numeraire" currency (such as USD), instead of one of X or Y. This is because traders and LPs usually denominate their profits, losses, and trade amounts in their native currency. Note, however, that this does not restrict one from studying the case where one of X and Y is the numeraire itself.

Definition 5.3.16 (LP's Belief)**.** *The belief of an LP is a function $\psi(\cdot,\cdot) : \mathbb{R}_+^2 \to \mathbb{R}_+$ such that it believes that at a future time, asset X will have price p_X (relative to the numeraire) and Y will have price p_Y with probability proportional to $\psi(p_X,\ p_Y)$.*

A belief function $\psi(\cdot,\cdot)$ has the following properties:

1. *ψ is integrable on any set of the form $\{p_X,\ p_Y \mid p_1 \leq p_X/p_Y \leq p_2\}$ for $p_1,\ p_2 \neq 0$.*

2. *There exists p_1, p_2 so that the integral of ψ on the set $\{p_X,\ p_Y \mid p_1 \leq p_X/p_Y \leq p_2\}$ is nonzero.*

3. *The set $\{p_X, p_Y \mid \psi(p_X, p_Y) > 0\}$ is open, and ψ is continuously differentiable on this set.*

4. *The integral of $\psi(p_X,\ p_Y)$ over its entire domain, $N_\psi = \iint_{p_X,\ p_Y} \psi(p_X, p_Y) dp_X\ dp_Y$, is a finite positive value.*

We do not normalize the belief function to integrate to 1 for ease of analysis later in the paper. This definition is strictly more flexible than a one-dimensional notion of a belief (i.e. a belief on the exchange rate between X and Y). A one-dimensional belief could be defined, for example, as nonzero only on the horizontal line where $p_Y = 1$ (with an appropriate adjustment to the notion of integrating over the belief). This flexibility will be important when we turn to the incentives of profit-seeking LPs (§5.6.2).

5.3.4 Belief Functions From Price Dynamics

An LP might not have just a belief about the distribution of future asset prices, but also some belief about how an asset's price will evolve over time. Applying time-discounting to beliefs about dynamics results in a belief distribution as in Definition 5.3.16.

Let $g(p_X, p_Y)$ be any continuous, integrable function of asset prices, and p_X^t and p_Y^t be stochastic processes that are believed to represent future asset price dynamics. Let $\rho_t(p_X, p_Y)$ be the joint probability density function at time t of p_X and p_Y induced by the stochastic processes. Denote the value of g at time t by g_t. The expected value of g_t is

$$\mathbb{E}(g_t) = \iint_{p_X, p_Y} g(p_X, p_Y)\rho_t(p_X, p_Y)\ dp_X\ dp_Y.$$

Denote the time-discounted value of g at the initial time, with discounting parameter γ by $g^{(\gamma)}$. By linearity of expectation, the expected value of $g^{(\gamma)}$ is

$$\begin{aligned}\mathbb{E}(g^{(\gamma)}) &= \int_{t=0}^{\infty} e^{-\gamma t} \left(\iint_{p_X, p_Y} g(p_X, p_Y)\rho_t(p_X, p_Y)\ dp_X\ dp_Y \right)\ dt \\ &= \iint_{p_X, p_Y} \int_{t=0}^{\infty} e^{-\gamma t}\ g(p_X, p_Y)\ \rho_t(p_X, p_Y)\ dt\ dp_X\ dp_Y\end{aligned}$$

Observe that $\mathbb{E}(g^{(\gamma)})$, where the expectation is over the price dynamics, is therefore equivalent to the expected value of g with respect to the static belief function $\psi(p_X, p_Y) = \int_{t=0}^{\infty} e^{-\gamma t}\ \rho_t(p_X, p_Y)\ dt$.

This holds for any integrable function g, which includes CFMM inefficiency (as in Proposition 5.4.1) but also expected profit and loss (as in §5.6).

This framework captures the geometric Brownian motion [297] model of price dynamics via:

$$\rho_t(p_X, p_Y) = \frac{1}{2\pi}\frac{1}{p_X p_Y \sigma_X \sigma_Y t}\exp\left(-\frac{\left(\ln p_X - \ln P_X - \left(\mu_X - \frac{1}{2}\sigma_X^2\right)t\right)^2}{2\sigma_X^2 t} - \frac{\left(\ln p_Y - \ln P_Y - \left(\mu_Y - \frac{1}{2}\sigma_Y^2\right)t\right)^2}{2\sigma_Y^2 t}\right).$$

Here, P_X and P_Y are the initial exchange rates of X and Y relative to the numeraire. μ_X and μ_Y are the drift parameters in the underlying Brownian motion of the log of p_X and p_Y. σ_X^2 and σ_Y^2 are the corresponding variances. With time discounting, this induces the following belief function.

$$\psi(p_X, p_Y) = \int_{t=0}^{\infty} e^{-\gamma t}\frac{1}{2\pi}\frac{1}{p_X p_Y \sigma_X \sigma_Y t}\exp\left(-\frac{\left(\ln p_X - \ln P_X - \left(\mu_X - \frac{1}{2}\sigma_X^2\right)t\right)^2}{2\sigma_X^2 t} - \frac{\left(\ln p_Y - \ln P_Y - \left(\mu_Y - \frac{1}{2}\sigma_Y^2\right)t\right)^2}{2\sigma_Y^2 t}\right)dt. \quad (5.1)$$

5.4 Optimizing for Liquidity Provision

How should LPs allocate capital to market-making at different exchange rates? This question is the core topic of our work. At any point in time, only the capital deployed near the reference exchange rate is useable for market-making. Thus, the “optimal” CFMM design necessarily depends on an LP’s belief on the distribution of future exchange rates.

We show here that an LP’s beliefs on future asset valuations can be compiled into an optimal CFMM design, which is the solution to a convex optimization problem (Theorem 5.4.4). Specifically, the optimization framework outputs a capital allocation $L(\cdot)$ (as in Definition 5.3.7) that minimizes the expected CFMM inefficiency (Proposition 5.4.1). Ultimately, we show that this relationship goes both ways; a liquidity allocation uniquely specifies an equivalence class of beliefs (Corollary 5.4.15). Per Observation 5.3.10, a liquidity allocation $L(\cdot)$ fully specifies a CFMM trading function.

This section discusses “optimality” from a viewpoint of minimizing CFMM inefficiency; however, we show in §5.6.1 that this optimization framework, with a different objective function, computes a CFMM that maximizes expected CFMM fee revenue. Furthermore, we show in §5.6.2 how to modify the objective of this program to account for losses incurred during CFMM operation.

5.4.1 A Convex Program for Optimal Liquidity Allocation

Objective: Minimize Expected CFMM Inefficiency

Proposition 5.4.1. *Suppose every trade order on a CFMM is for one unit numeraire’s worth of either X or Y, and buys or sells the asset in question with equal probability. The expected CFMM inefficiency is $\frac{1}{N_\psi}\iint_{p_X, p_Y}\frac{\psi(p_X, p_Y)}{p_Y L(p_X/p_Y)}dp_X\ dp_Y$. We define the integral only where $\psi(p_X, p_Y) > 0$.*

Further, we define $\psi(p_X, p_Y)/L(p_X/p_Y)$ to be ∞ when $L(p_X/p_Y) = 0$. N_ψ is as in Definition 5.3.16.

Proof. Suppose that a trader order is for 1 unit of numeraire's worth of X with probability α, and for 1 unit of numeraire's worth of Y with probability $1-\alpha$. The size of a trade denominated in X is therefore $1/p_X$, and the size of a trade denominated in Y is $1/p_Y$.

Recall from Definition 5.3.15 that at a given set of reference prices p_X, p_Y, the CFMM inefficiency for a trade buying or selling 1 numeraire's worth of X is $\frac{\hat{p}}{p_X}\frac{1}{L(p_X/p_Y)} = \frac{p_X}{p_Y p_X}\frac{1}{L(p_X/p_Y)} = \frac{1}{p_Y L(p_X/p_Y)}$. Similarly, also from Definition 5.3.15, the CFMM inefficiency corresponding to a trade of 1 numeraire's worth of Y is $\frac{1}{p_Y}\frac{1}{L(p_X/p_Y)}$. Hence, the overall expected CFMM inefficiency is

$$\frac{1}{N_\psi}\iint_{p_X,p_Y}\psi(p_X,p_Y)\left(\frac{\alpha}{p_Y L(p_X/p_Y)} + \frac{1-\alpha}{p_Y L(p_X/p_Y)}\right) dp_X\ dp_Y,$$

$$= \frac{1}{N_\psi}\iint_{p_X,p_Y}\frac{\psi(p_X,p_Y)}{p_Y L(p_X/p_Y)} dp_X\ dp_Y. \qquad \square \qquad (5.2)$$

For clarity of exposition, we focus on the scenario where each order trades 1 unit of the numeraire's worth of value. Our model can study, however, scenarios where for general trade sizes and also when the trade size is a function of p_X and p_Y. The CFMM inefficiency is a linear function of trade size. A distribution of trade sizes can be multiplied with the belief function.

Proposition 5.4.1 also implies that the trade failure chance is the same for a trader buying X or Y. The p_Y in the denominator of the integrand in equation (5.2) appears because the liquidity $L(\cdot)$ is defined with respect to the reserves of asset Y, i.e., $\mathcal{Y}(\cdot)$ (recall Lemma 5.3.9). Overall, there is no distinction between X and Y for the purpose of the CFMM inefficiency.

Constraints: A Finite Budget for Market-Making

The asset reserves of a CFMM are finite. Clearly, the best CFMM to minimize expected inefficiency has liquidity $L(p) = \infty$ at every exchange rate p, but this would require an infinite amount of each asset (Observation 5.3.10). We model an LP with a fixed budget B who creates a CFMM when the reference exchange rates of X and Y in the numeraire are P_X and P_Y, respectively. With this budget, the LP can purchase (or borrow) any amount of X and Y, say, X_0 and Y_0, subject to the constraint that $P_X X_0 + P_Y Y_0 \leq B$. With this intuition, we have the following technical lemmas:

Lemma 5.4.2. *Given a purchasing choice of X_0 and Y_0, the LP can choose $L(\cdot)$ and set the initial spot exchange rate of the CFMM to be p_0, subject to the following asset conservation constraints.*

1. $\int_0^{p_0} \frac{L(p)}{p} dp \leq Y_0$

2. $\int_{p_0}^{\infty} \frac{L(p)}{p^2} dp \leq X_0$

Proof. Follows from Observation 5.3.10. □

Lemma 5.4.3. *For any two budgets B, B' with $B' > B$ and any capital allocation $L_1(\cdot)$ satisfying the constraints of Lemma 5.4.2 with budget B, there exists a capital allocation $L_2(\cdot)$ satisfying the constraints of Lemma 5.4.2 using the larger budget B' that gives a strictly lower expected CFMM inefficiency.*

Proof. Duplicate $L_1(\cdot)$ and allocate the capital $B' - B$ to any p with $\psi(p, 1) > 0$ to build $L_2(\cdot)$. □

A rational LP sets the initial spot exchange rate of the CFMM to be equal to the current reference exchange rate (i.e. $p_0 = \frac{P_X}{P_Y}$). If not, a trader could arbitrage the CFMM against an external market. The arbitrage profit of this trader is the LP's loss, which effectively reduces the LP's initial budget.

Our convex program combines the above objective and constraints to compute an optimal liquidity allocation $L(p)$. The core of the rest of this work is in using this program to understand the relationship between LP beliefs and optimal liquidity allocations.

Theorem 5.4.4. *Suppose that the initial reference prices of assets X and Y are P_X and P_Y, and that an LP has initial budget $B > 0$ and belief function $\psi(\cdot, \cdot)$.*

The optimal liquidity provision strategy, $L(\cdot)$, is the solution to the following convex optimization problem (COP). The decision variables are X_0, Y_0, and $L(p)$ for each exchange rate $p > 0$. [5]

$$\textit{minimize} \quad \iint_{p_X, p_Y} \frac{\psi(p_X, p_Y)}{p_Y L(p_X/p_Y)} dp_X \; dp_Y \tag{COP}$$

$$\textit{subject to} \quad \int_0^{p_0} \frac{L(p)}{p} dp \leq Y_0 \tag{COP1}$$

$$\int_{p_0}^{\infty} \frac{L(p)}{p^2} dp \leq X_0 \tag{COP2}$$

$$X_0 P_X + Y_0 P_Y \leq B \tag{COP3}$$

$$L(p) \geq 0 \qquad \forall\, p > 0 \tag{COP4}$$

Proof. The $L(\cdot)$ that solves COP minimizes the expected transaction failure chance (the expression in Proposition 5.4.1),[6] while satisfying the LP's budget constraint. The objective and the constraints are integrals of convex functions and thus are convex.

This optimization problem is over a Banach space (there are uncountably many $L(p)$). Well-established results from the theory of optimization over Banach spaces show that optimal solutions

[5]The optimization is over a Banach space with one dimension for each $p > 0$; we elide this technicality when possible for clarity of exposition.

[6]The normalization term in the denominator is dropped for clarity since it doesn't change the solution of the problem.

exist (Theorem 47.C, [318]) and the KKT conditions are well defined (§4.14, Proposition 1, [319]). □

A CFMM offers only a spot exchange rate (X relative to Y), not a spot valuation for each asset (relative to the numeraire). In this light, we find that the objective function of COP can be rearranged to one that depends only on ratios of valuations.

Lemma 5.4.5. *Define r, θ to be the standard polar coordinates, with $p_X = r\cos(\theta)$ and $p_Y = r\sin(\theta)$.*

$$\iint_{p_X,p_Y} \frac{\psi(p_X,p_Y)}{p_Y L(p_X/p_Y)} dp_X \ dp_Y = \int_\theta \left(\frac{1}{L(\cot(\theta))\sin(\theta)} \int_r \psi(r\cos(\theta), r\sin(\theta)) dr \right) d\theta$$

Proof. Follows by standard algebraic manipulations ($dp_X \ dp_Y = r \ dr \ d\theta$). □

This rearrangement reveals a useful equivalence class among LP beliefs.

Corollary 5.4.6. *Any two beliefs ψ_1, ψ_2 give the same optimal liquidity allocations if there exists a constant $\alpha > 0$ such that for every θ,*

$$\int_r \psi_1(r\cos(\theta), r\sin(\theta))dr = \alpha \int_r \psi_2(r\cos(\theta), r\sin(\theta))dr$$

This corollary has important implications for the closed-form results we obtain in §5.5 for commonly deployed CFMMs. The analysis of a belief defined on the square $p_X, p_Y \in (0, P_X] \times (0, P_Y]$ gives the results for all beliefs defined analogously on $p_X, p_Y \in (0, \alpha P_X] \times (0, \alpha P_Y]$ for any $\alpha > 0$.

Corollary 5.4.7. *Define $\varphi_\psi(\theta) = \int_r \psi(r\cos(\theta), r\sin(\theta))dr$. Then*

$$\iint_{p_X,p_Y} \frac{\psi(p_X,p_Y)}{p_Y L(p_X/p_Y)} dp_X \ dp_Y = \int_p \frac{\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}{L(p)} dp$$

Corollary 5.4.7 enables a straightforward construction of a feasible solution to COP.

Lemma 5.4.8. *COP always has a solution with finite objective value.*

Corollary 5.4.9. *On any set of nonzero measure, we cannot have $\psi(p_X, p_Y) > 0$ and $L(p_X/p_Y) = 0$.*

Proofs of Corollarys 5.4.6 and 5.4.7 and Lemma 5.4.8 are in the Appendix C.3.2, C.3.3, and C.3.4 respectively.

5.4.2 Optimality Conditions

We first give some lemmas about the structure of optimal solutions to COP.

Lemma 5.4.10. *The following hold at any optimal solution.*

1. $\int_0^{p_0} \frac{L(p)}{p} dp = Y_0$

2. $\int_{p_0}^{\infty} \frac{L(p)}{p^2} dp = X_0$

3. $X_0 P_X + Y_0 P_Y = B$

Lemma 5.4.10 says that at optimum, the constraints of COP are tight. A full proof is in Appendix C.3.5. Using the result of Lemma 5.4.10, the KKT conditions (§5.5.3, [99]) of COP are the following:

Lemma 5.4.11 (KKT Conditions)**.** *Let λ_Y, λ_X, and λ_B be the Lagrange multipliers for COP1, COP2, and COP3 respectively. Let $\{\lambda_{L(p)}\}$ be the Lagrange multipliers for each $L(p) \geq 0$ constraint.*

When $\varphi_\psi(\cot^{-1}(p)) > 0$:

1. *For all p with $p \geq p_0$, $\frac{\lambda_X}{p^2} = \frac{1}{L(p)^2} \varphi_\psi(\cot^{-1}(p)) \sin(\cot^{-1}(p)) + \lambda_{L(p)}$.*

2. *For all p with $p \leq p_0$, $\frac{\lambda_Y}{p} = \frac{1}{L(p)^2} \varphi_\psi(\cot^{-1}(p)) \sin(\cot^{-1}(p)) + \lambda_{L(p)}$.*

3. *$\lambda_X = P_X \lambda_B$ and $\lambda_Y = P_Y \lambda_B$.*

When $\varphi_\psi(\cot^{-1}(p)) = 0$:

1. *For all p with $p \geq p_0$, $\frac{\lambda_X}{p^2} = \lambda_{L(p)}$.*

2. *For all p with $p \leq p_0$, $\frac{\lambda_Y}{p} = \lambda_{L(p)}$.*

3. *$\lambda_X = P_X \lambda_B$ and $\lambda_Y = P_Y \lambda_B$.*

Proof. These are the KKT conditions of COP. $\{L(p)\}$ is a functional over a Banach space. This functional exists for every optimal solution by Proposition 1 of §4.14 of [319]. Note that that proposition requires the objective to be continuously differentiable in a neighborhood of the optimal solution; this does not hold when the optimization problem is as written and there is some p so that $\varphi_\psi(\cot^{-1}(p))$ goes continuously to 0 at p (but is nonzero near p). In this case, one could replace $L(p)$ by $L(p) + \varepsilon$ in the denominator of the objective, for some arbitrarily small ε. This would cause a small distortion in $L(p)$. We elide this technicality for clarity of exposition. Continuous differentiability of the objective on a neighborhood where $L(p) > 0$ for all p with $\varphi_\psi(\cot^{-1}(p)) > 0$ follows from the assumption that ψ is continuously differentiable on the set where $\psi(p_X, p_Y) > 0$, and that this set is open (in Definition 5.3.16). □

Corollary 5.4.12. *The integral $\mathcal{Y}(\tilde{p}) = \int_0^{\tilde{p}} \frac{L(p)dp}{p}$ is well defined for every $\tilde{p}$ and $\mathcal{Y}(\cdot)$ is monotone nondecreasing and continuous.*

A proof is given in Appendix C.3.6. Lemma 5.4.11 and Corollary 5.4.12 together imply that the behavior of a CFMM that results from an optimal solution of COP is well-defined.

Consequences of KKT Conditions

The KKT conditions immediately imply the following facts about any optimal solution of COP.

Lemma 5.4.13.

1. $\lambda_Y Y_0 = \int_0^{p_0} \frac{\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}{L(p)}dp$ *and* $\lambda_X X_0 = \int_{p_0}^{\infty} \frac{\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}{L(p)}dp$.

2. $Y_0 > 0$ *implies* $\lambda_Y > 0$. *Similarly,* $X_0 > 0$ *implies* $\lambda_X > 0$.

3. $L(p) \neq 0$ *if and only if* $\lambda_{L(p)} = 0$ *(unless, for* $p \leq p_0$*,* $\lambda_Y = 0$ *or for* $p \geq p_0$*,* $\lambda_X = 0$*).*

4. *The objective value is* $\lambda_Y Y_0 + \lambda_X X_0$.

5. $\frac{\lambda_X}{P_X} = \frac{\lambda_Y}{P_Y}$.

Proof.

1. Multiply each side of the first KKT condition in Lemma 5.4.11 by $L(p)$ (for p with nonzero $\varphi_\psi(\cot^{-1}(p))$ to get $\frac{\lambda_X L(p)}{p^2} = \frac{1}{L(p)}\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p)))$, integrate from p_0 to ∞, and apply the second item of Lemma 5.4.10.

   A similar argument (integrating from 0 to p_0) gives the expression on $\lambda_Y Y_0$.

2. If $Y_0 > 0$, then the right side of the equation in the previous part is nonzero, so λ_Y must be nonzero. The case of λ_X is identical.

3. Follows from points 1 and 2 of Lemma 5.4.11.

4. The right sides of the equations in the first statement add up to the objective.

5. Follows from point 3 of Lemma 5.4.11 □

Lemma 5.4.13 shows that the fraction of liquidity allocated to an exchange rate p is a function only of the LP's (relative) belief that the future exchange rate will be p. Specifically, except through an overall scalar, there is no interaction between the values of $L(\cdot)$ at different relative exchange rates.

Proposition 5.4.14. *At an optimum,* $L(p)$ *is a function of* $\lambda_X, \lambda_Y, \varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))$*, and* p.

Proof. Follows from Lemma 5.4.11. □

Proposition 5.4.14 gives several important consequences. First, it shows that an optimal liquidity allocation can be inverted to give a set of belief functions that lead to that liquidity allocation.

Corollary 5.4.15. *A liquidity allocation $L(\cdot)$ and an initial spot exchange rate p_0 are sufficient to uniquely specify an equivalence class of beliefs (as defined in Corollary 5.4.6) for which $L(\cdot)$ is optimal.*

Second, Proposition 5.4.14 actually enables an explicit construction of a belief that leads to $L(\cdot)$.

Corollary 5.4.16. *Let P_X and P_Y be initial reference valuations, and let $L(\cdot)$ denote a liquidity allocation. Define the belief $\psi(p_X, p_Y)$ to be $\frac{(L(p_X/p_Y))^2}{p_X/p_Y}$ when $p_X \in (0, P_X]$ and $p_Y \in (0, P_Y]$, and to be 0 otherwise. Then $L(\cdot)$ is the optimal allocation for $\psi(\cdot, \cdot)$.*

Finally, the KKT conditions (Lemma 5.4.11) imply that linear combinations of beliefs result in predictable combinations of liquidity allocations. Towards this, we have the following result, which will be useful in further proofs. We prove these corollaries in Appendices §C.3.7, C.3.8, and C.3.9.

Corollary 5.4.17. *Let ψ_1, ψ_2 be any two belief functions (that give φ_{ψ_1} and φ_{ψ_2}) with optimal allocations $L_1(\cdot)$ and $L_2(\cdot)$, and let $L(\cdot)$ be the optimal allocation for $\psi_1 + \psi_2$. Then $L^2(\cdot)$ is a linear combination of $L_1^2(\cdot)$ and $L_2^2(\cdot)$.*

Further, when φ_{ψ_1} and φ_{ψ_2} have disjoint support, $L(\cdot)$ is a linear combination of $L_1(\cdot)$ and $L_2(\cdot)$.

5.5 Common CFMMs and Beliefs

We turn now to the CFMMs deployed in practice. What do the choices of trading functions in large CFMMs reveal about practitioners' beliefs about future asset prices? In fact, the optimal beliefs for several widely-used trading functions closely match the widespread but informal intuition about these systems. Recall that P_X and P_Y are the initial reference exchange rates. Also, recall the assumption that all trades are for the worth of 1 unit of the base numeraire currency (Proposition 5.4.1).

5.5.1 The Uniform, Independent Belief: Constant Product Market Makers

Proposition 5.5.1. *Let $\psi(p_X, p_Y) = 1$ on $(0, P_X] \times (0, P_Y]$ and 0 otherwise. The liquidity allocation $L(\cdot)$ that minimizes the CFMM inefficiency is the allocation implied by the trading function $f(x, y) = xy$.*

Of course, by Corollary 5.4.6, the belief that gives the constant product market maker is not unique. Importantly, rescaling the belief to one defined analogously on the rectangle $(0, \alpha P_X] \times (0, \alpha P_Y]$ for any constant $\alpha > 0$ does not change the optimal liquidity allocation. This invariance of the optimal liquidity allocation to such transformations of the belief applies to all results in this section.

Proof. For the CFMM $f(x, y) = xy$, we have $\mathcal{X}(p)\mathcal{Y}(p) = X_0Y_0$ and $p = \mathcal{Y}(p)/\mathcal{X}(p)$. This implies $\mathcal{Y}(p)^2 = pX_0Y_0$ and $\mathcal{Y}(p) = \sqrt{pX_0Y_0}$. Recall that $L(p) = \frac{d\mathcal{Y}(p)}{d\ln(p)}$. This gives $L(p) = \frac{\sqrt{pX_0Y_0}}{2}$.

Corollary 5.4.16 shows that a belief that leads to this liquidity allocation is $\frac{X_0Y_0}{4}$ on the rectangle $(0, P_X] \times (0, P_Y]$ and 0 elsewhere. The result follows by rescaling the belief (Corollary 5.4.6). □

Proposition 5.5.1 captures the folklore intuition within Decentralized Finance regarding the circumstances in which constant product market makers are optimal. If an LP has no information regarding correlations in the reference valuations of assets (in terms of the numeraire), then the LP should choose one of these CFMMs because it allocates liquidity evenly across the entire range of exchange rates. To be specific, the liquidity available at a given exchange rate for purchasing Y from the CFMM is always proportional to the amount of Y in the reserves at that exchange rate.

5.5.2 Uniform Beliefs on Exchange Rate Ranges: Concentrated Liquidity Positions

Some CFMMs (e.g. [56]) allow LPs to create piecewise-defined trading strategies, often called "concentrated liquidity CFMMs."

Definition 5.5.2 (Concentrated Liquidity CFMM)**.** *A trading function $f'(x, y) = f(x + \hat{x}, y + \hat{y})$ for some constants $\hat{x} > 0, \hat{y} > 0$, has nonzero liquidity on a smaller range of exchange rates than f.*

While a "concentrated liquidity CFMM" f' can be designed with any f, we focus on those which use $f(x, y) = xy$ since these have been widely adopted in practice after being introduced by [56].

Observe that this trading function differs from the constant product trading rule when x or y reaches 0. There exists a range of exchange rates $(p_{\min}, p_{\max})$ on which this CFMM makes the same trades as one based on the constant product rule. Outside of this range, the CFMM makes no trades. There is a direct mapping from $(\hat{x}, \hat{y})$ to $(p_{\min}, p_{\max})$, we omit it here for clarity of exposition.

This trading function corresponds to a belief pattern that is restricted in a similar way; on the specified range of exchange rates, the belief is the same as that of Proposition 5.5.1, and 0 otherwise.

Proposition 5.5.3. *Let $p_{min} < p_{max}$ be two arbitrary exchange rates, and let $\psi(p_X, p_Y) = 1$ if and only if $0 \leq p_X \leq P_X$, $0 \leq p_Y \leq P_Y$, and $p_{min} \leq p_X/p_Y \leq p_{max}$, and 0 otherwise. The allocation $L(\cdot)$ that maximizes the fraction of successful trades is the allocation implied by a concentrated liquidity position with price range defined by p_{min} and p_{max}.*

A proof is given in Appendix §C.3.10. LPs who make markets using concentrated liquidity CFMMs implicitly expect that while the valuations of the two assets may move up and down, their movements are correlated; that is, the exchange rate always stays within some range. This belief exactly matches that intuition.

Corollary 5.5.4. *The belief corresponding to multiple concentrated liquidity positions, which are defined on disjoint ranges of exchange rates, is a linear combination of the beliefs that correspond to the individual concentrated liquidity positions (as specified in Proposition 5.5.3).*

Proof. Application of Corollary 5.4.17. □

5.5.3 Skew in Belief Function: Weighted Product Market Makers

Weighted Constant Product Market Makers add weights to a constant product curve to get trading functions of the form $f(x, y) = x^\alpha y$, for some constant $\alpha > 0$.

Proposition 5.5.5. *The belief function* $\psi(p_X, p_Y) = \left(\frac{p_X}{p_Y}\right)^{\frac{\alpha-1}{\alpha+1}}$ *when* $(p_X, p_Y) \in (0, P_X] \times (0, P_Y]$ *and* 0 *otherwise corresponds to the weighted product market maker* $f(x, y) = x^\alpha y$.

A proof is given in Appendix C.3.13. This proposition shows that LPs who use weighted product market makers expect that the value of one asset will typically be much higher than that of the other. Informally, more liquidity is allocated towards the higher ranges of exchange rates than the lower ranges when $\alpha > 1$ (and vice versa for $\alpha < 1$). This CFMM, therefore, can satisfy a higher fraction of trades when the exchange rate is high (resp. low). On the other hand, a skewed allocation means that some price ranges suffer from much higher slippage for a fixed-size trade. This intuition mirrors the description of how LPs should choose the weight α [81] in a public deployment [241] of weighted product market makers. This theorem reduces to Proposition 5.5.1 when $\alpha = 1$.

5.5.4 Logarithmic Market Scoring Rule

The logarithmic market scoring rule (LMSR) [191], which has been used extensively in the context of prediction markets, corresponds to a CFMM with trading function $f(x, y) = 2 - e^{-x} - e^{-y}$ [273].

Proposition 5.5.6. *The optimal trading function to minimize the expected CFMM inefficiency for the belief* $\psi(p_X, p_Y) = \frac{p_X p_Y}{(p_X + p_Y)^2}$ *when* $(p_X, p_Y) \in (0, P_X] \times (0, P_Y]$ *and* 0 *otherwise, is* $f(x, y) = 2 - e^{-x} - e^{-y}$.

A proof is given in Appendix C.3.12. Observe that ψ is symmetric about the line $p_Y = p_X$ — that is to say, $\psi(p_X, p_Y) = \psi(p_Y, p_X)$. We analyse this belief function in polar coordinates to get a better intuition. Note the term $\frac{r\cos(\theta) r\sin(\theta)}{(r\cos(\theta)+r\sin(\theta))^2} = \frac{\sin(2\theta)}{2(1+\sin(2\theta))}$, which is maximized at $\theta = \pi/4$. For initial exchange rates with $P_X = P_Y$, the LMSR expects the relative exchange rate to concentrate about $\frac{p_X}{p_Y} = 1$. At extreme exchange rates ($\frac{p_X}{p_Y} \to 0$ or $\frac{p_X}{p_Y} \to \infty$), the belief goes to 0. The LMSR-based CFMM correspondingly allocates very little liquidity at extreme exchange rates.

5.6 Market-Maker Profit and Loss

Deploying assets within a CFMM has a cost, and LPs naturally want to understand the financial tradeoffs involved (instead of just minimising CFMM inefficiency). Specifically, LPs in CFMMs trade off revenue from transaction fees against losses due to adverse selection.

5.6.1 Fee Revenue

Many CFMM deployments charge a fixed-rate fee on every transaction. While some early instantiations automatically reinvested fee revenue within the CFMM reserves [55], more recent deployments choose not to [56]. We consider the case where fees are not automatically reinvested and, for simplicity, the case where fees are immediately converted into the numeraire currency.

Observe that a percentage-based fee on a trade is, from the trader's point of view, equivalent to a multiplicative factor in the exchange rate. Slippage also measures the deviation of a trader's received exchange rate from the reference exchange rate. A transaction fee, therefore, has the same effect in our model on a CFMM's ability to settle trades as a reduction in a user's tolerated slippage.

The fee revenue depends on the rate of trade requests and the fraction of trades the CFMM can settle. Recall the probability q of a trade request arriving at a time step from the system model in Definition 5.3.1 and the trade size distribution $size(\cdot)$ from the trade model in Definition 5.3.5. These can be compiled into a belief on the "rate" of trade requests that the LP expects the CFMM will see.

Definition 5.6.1 (Transaction Rate Model)**.** *Denote as $rate_\delta(p_X, p_Y)$ denote an LP's prediction on the expected volume of trades (in terms of the numeraire) attempted on the CFMM when the reference prices are p_X and p_Y and the trading fee is set to δ.*

Proposition 5.6.2. *The expected revenue of a CFMM in one unit of time, using transaction fee δ, when the mean trade-size is worth s units of the numeraire, is*

$$\mu(rate, \psi, \delta, s) = \frac{\delta}{N_\psi} \iint_{p_X, p_Y} rate_\delta(p_X, p_Y)\psi(p_X, p_Y) \left(1 - \frac{s}{p_Y L(p_X/p_Y)}\right) dp_X \; dp_Y.$$

Proof. The expected revenue of the CFMM is exactly the transaction fee times the volume of trading that goes through the CFMM, which is equal to the predicted input trade volume, less the number of trades that the CFMM cannot settle. □

In any real-world setting, the fee δ influences the predilection of traders to use the CFMM. However, the transaction fee is a predetermined constant in many of the most widely used CFMM deployments. Uniswap V2, for example, sets a fixed 0.3% fee [55], and Uniswap V3 lets LPs choose between three choices of fee rates [56]. In this model, LPs would consider each fee rate they are allowed to set (or perform a grid search over many fee schedules), and then predict (through some external knowledge, outside the scope of this work) the transaction rate at that fee schedule.

Corollary 5.6.3. *The allocation $L(\cdot)$ that maximizes fee revenue is same as the $L(\cdot)$ that minimizes*

$$\iint_{p_X,p_Y} \frac{rate_\delta(p_X,p_Y)\psi(p_X,p_Y)}{p_Y L(p_X/p_Y)} dP_X \ dP_Y$$

which is in turn equal to the liquidity allocation $L(\cdot)$ which is optimal for an LP with belief $rate_\delta(p_X,p_Y) *$ $\psi(p_X,p_Y)$ and is concerned only with minimizing CFMM inefficiency.

In simpler terms, an expected distribution on future transaction rates is equivalent, in the eyes of the optimization framework, to a belief on future prices. A revenue-maximizing LP can therefore use the same optimization problem (with an adjusted input) as in Theorem 5.4.4.

Definition 5.6.1 assumes traders are attempting trades for exogenous reasons. However, in the real world, trade volume might depend on how effectively a market-maker provides liquidity. Note that when the trade input rate is independent of p_X and p_Y, then the objective of minimizing CFMM inefficiency produces the same liquidity allocation as an objective of maximizing the fee collected.

5.6.2 Liquidity Provider Loss

LPs may also suffer losses when asset prices change. As discussed in Milionis et al. [244] and Cartea [112], these losses come from two sources: first, from the asset price movements directly (exposure to market-risk), and second, from shifts in the relative exchange rate between the assets. As the relative exchange rate on the external market shifts, arbitrageurs can trade with the CFMM to realign the CFMM's spot exchange rate with the external market's exchange rate, making a profit in the process at the CFMM's expense. Milionis et al. [244] show that the expected lose due to arbitrage is higher for CFMMs trading more volatile assets.

The CFMM's loss can only be defined relative to a counterfactual choice; instead of engaging in market-making, an LP could deploy their capital in some other manner. For example, a simple counterfactual would be to hold a fixed amount of the numeraire or a fixed quantity of each asset and, at a future time, compare the value of this strategy with the value of the CFMM's asset reserves.

Proposition 5.6.4. *The expected future value of the CFMM's reserves, as per belief $\psi(p_X,p_Y)$, is*

$$\nu(\psi) = \frac{1}{N_\psi} \iint\limits_{p_X,p_Y} \psi(p_X,p_Y)\left(p_X\mathcal{X}(p_X/p_Y) + p_Y\mathcal{Y}(p_X/p_Y)\right) dp_X \ dp_Y$$

$$= \frac{1}{N_\psi}\int\limits_0^\infty \left(\frac{L(p)}{p^2} \iint\limits_{p_X,p_Y} p_X\psi(p_X,p_Y)\mathbb{1}_{\{\frac{p_X}{p_Y}\leq p\}} dp_X dp_Y + \frac{L(p)}{p} \iint\limits_{p_X,p_Y} p_Y\psi(p_X,p_Y)\mathbb{1}_{\{\frac{p_X}{p_Y}\geq p\}} dp_X dp_Y \right) dp$$

where $\mathcal{X}(p)$ and $\mathcal{Y}(p)$ denote the amounts of X and Y held in the reserves at spot exchange rate p, and $\mathbb{1}_E$ is the characteristic function of the event E.

This expression for $\nu(\psi)$ is a linear function of each $L(p)$.

Proposition 5.6.4 follows from representing $\mathcal{X}(p_X/p_Y)$ and $\mathcal{Y}(p_X/p_Y)$ in terms of integration of $L(p)$ per Observation 5.3.10 and then changing the order of integration. A full proof is in Appendix C.3.13.

Divergence Loss

Observe that the expected value of a “buy-and-hold" counterfactual strategy does not depend on a chosen allocation $L(\cdot)$. In this case, the expected loss of the market maker (relative to a counterfactual strategy with expected payoff C) is $C - \nu(\psi)$ and is linear in each $L(p)$. This measurement of loss is typically called “divergence loss”.

Loss-Vs-Rebalancing

Alternatively, Milionis et al. [244] propose the “LVR” metric that compares a CFMM's performance against that of a counterfactual strategy that makes the same trades in asset X as a CFMM but at the reference exchange rate, not at the CFMM's spot rate. This LVR is an expectation over a model of price dynamics but depends only on that model and the so-called “instantaneous LVR”, which is proportional to $-p^2\frac{\mathcal{X}(p)}{dp} = L(p)$ (Theorem 1 and Lemma 1, [244]). Milionis et al. [244] show that the “instantaneous LVR” is equivalent to the loss to the CFMM due to arbitrage.

The overall LVR can be given by integrating the “instantaneous LVR” over the time period for which the LP operates the CFMM. Since the overall LVR is, therefore, a linear function of $L(\cdot)$, it can be incorporated directly into our optimization framework.

5.6.3 Net Profit

Adding capital to a CFMM is profitable in expectation if and only if the expected fee revenue outweighs the expected loss. In the following, we analyze the net expected profit of an LP.

Lemma 5.6.5. *Let $\mu(\cdot)$ be as defined in Proposition 5.6.2 and $\Gamma(\cdot)$ be loss function which is linear in $L(p)$ for each p. An LP's net expected profit, $\mu(rate, \psi, \delta, s) - \Gamma(L(\cdot))$, is a concave function in $L(p)$ for each p.*

Proof. $\mu(\cdot)$ is clearly concave, and for any loss function which is linear in each variable $L(p)$, the whole function is concave in each $L(p)$. $\square$

Crucially, Lemma 5.6.5 is applicable to both the divergence loss and the Loss-vs-Rebalancing.

Observation 5.6.6. *A profit-maximizing liquidity allocation by maximizing $\mu(rate, \psi, \delta, s) - \Gamma(L(\cdot))$, and is computable via a convex optimization problem.*

The objective is the expression of Lemma 5.6.5, and the constraints are the same as in Theorem 5.4.4.

Not only, therefore, does our optimization framework allow an LP to design a CFMM that maximizes successful trading activity, but it can also guide a profit-maximizing LP to a profit-maximizing CFMM. Observe that the objective value of this convex program is positive precisely when the CFMM generates a profit.

5.6.4 Divergence Loss Shifts Liquidity Away From the Current Exchange Rate

The threat of divergence loss is in inherent tension with the potential of a CFMM to earn fee revenue by settling trades. A CFMM can, of course, vacuously eliminate divergence loss by refusing to make any trades (by allocating no liquidity to any price), and in general, higher liquidity at a given range of exchange rates leads to better CFMM performance at that range and higher divergence loss if the reference exchange rate moves away from that range.

When divergence loss is accounted for in the objective, our optimization framework computes a liquidity allocation that trades off fee revenue with divergence loss by shifting liquidity away from the current exchange rate i.e., towards more extreme exchange rates (closer to $p = 0$ and $p = \infty$).

This qualitative behavior is, however, not true of every possible loss function that an LP might add to the objective. Each function will have its own effect; nevertheless, the optimization problem can always be solved to compute an optimal allocation.

Theorem 5.6.7. *Let $L_1(p)$ be the optimal liquidity allocation that maximizes fee revenue — the solution to the optimization problem for the objective of minimizing the following:*

$$-\frac{\delta}{N_\psi}\iint_{p_X,p_Y} rate_\delta(p_X,p_Y)\psi(p_X,p_Y)\left(1-\frac{s}{p_Y L(p_X/p_Y)}\right)dp_X\ dp_Y$$

Let $L_2(p)$ be the optimal liquidity allocation that maximizes fee revenue while accounting for divergence loss — the solution to the optimization problem for the objective of minimizing the following:

$$-\nu(\psi)-\frac{\delta}{N_\psi}\iint_{p_X,p_Y} rate_\delta(p_X,p_Y)\psi(p_X,p_Y)\left(1-\frac{s}{p_Y L(p_X/p_Y)}\right)dp_X\ dp_Y$$

Let $X_1 = \int_{p_0}^{\infty}\frac{L_1(p)}{p^2}dp$ and $X_2 = \int_{p_0}^{\infty}\frac{L_2(p)}{p^2}dp$ be the optimal initial quantities of X for the above two problems respectively.

Then there exists some $p_1 > p_0$ such that for $p_0 \le p \le p_1$, $\frac{L_1(p)}{X_1} \ge \frac{L_2(p)}{X_2}$ and for $p \ge p_1$, $\frac{L_1(p)}{X_1} \le \frac{L_2(p)}{X_2}$. An analogous statement holds for the allocations of Y.

The proof is technical and is given in Appendix C.3.13. In simple terms, the divergence loss might change the optimal initial choice of reserves, but given that choice, divergence loss causes liquidity to shift away from the current exchange rate. Qualitatively, the higher the magnitude of the expected divergence loss relative to the expected fee revenue, the larger the magnitude of this effect. This reflects the well-known intuition that when the reference exchange rate changes substantially, highly concentrated liquidity positions (like in §5.5.2) suffer higher losses than more evenly distributed liquidity allocations (like those of §5.5.1).

5.7 Conclusion

We develop in this work a convex program that, for any set of beliefs about future asset valuations, outputs a trading function that maximizes the expected fraction of trade requests the CFMM can settle. Careful analysis of this program allows for study of the inverse relationship as well; for any trading function, our program can construct a class of beliefs for which the trading function is optimal. Constructing this program requires a new notion of the liquidity of a CFMM, and the core technical challenge involves careful analysis of the KKT conditions of this program.

Unlike prior work, this program is able to explain the diversity of CFMM trading curves observed in practice. We analyze several CFMMs that are widely deployed in the modern DeFi ecosystem, and show that the beliefs revealed by our model match the informal intuition of practitioners.

Furthermore, our program is versatile enough to cover the case of a profit-seeking LP that must trade off expected revenue from trading fees against loss due to arbitrage. This program therefore can serve as a guide for practitioners when choosing a liquidity allocation in a real-world CFMM.

Chapter 6

Augmenting Batch Exchanges with Constant Function Market Makers

6.1 Introduction

A crucial component of an economic system is a structure to facilitate the exchange of assets.

Recall that this thesis, thus far, has discussed two separate mechanisms for implementing an exchange, but each operates in a different computational model and with different economic tradeoffs. Chapter 3 introduced a linearly-scalable instance of a batch exchange, trading between many assets simultaneously, in the computational model—using unordered blocks of commutative transactions—of chapter 2. By contrast, chapter 5 studies the tradeoffs inherent in an automated exchange mechanism known as the Constant Function Market Maker (CFMM), which sequentially executes one trade after another.

These mechanisms have not only different computational tradeoffs but also distict economic characteristics. Some models of batch auctions suggest that they can lead to improved price discovery or reduce intermediation costs [236, 129, 155, 284], at the cost of additional trade latency. Budish et. al [103, 73] argue that batch auctions address the problem of "stale quote sniping" in traditional, centralized exchanges [1], reduce the cost of providing liquidity, and move competition from speed to price. Others argue the opposite; in some contexts, batch auctions could reduce market liquidity [246, 219, 153].

Observe that the replicated state machine paradigm (as in §1.2.1) already incurs latency in each transaction, so SPEEDEX incurs no additional latency. Furthermore, because public blockchains

[1] A stale quote is an offer to buy or sell an asset whose limit price is outdated (perhaps because the rest of the market has responded to new information). A market-maker may attempt to cancel the offer, but a trader with a lower-latency connection to the exchange may trade with ("snipe") the outdated offer first.

batch transactions for consensus, and then order transactions within each batch, the order of transactions can be explicitly manipulated for market activity, such as direct front-running [143] that far exceeds stale quote sniping and is usually heavily curtailed by regulation in traditional exchanges (i.e. [240, 42]). Batch auctions eliminate this class of front-running, because every trade in a batch happens at the same price. Additionally, exchanges like SPEEDEX that trade many assets in a single batch sidestep the problem of liquidity fragmentation between different trading pairs, which is especially problematic in modern blockchains [220], at the cost of substantially complicating the process of computing batch clearing prices.

By contrast, chapter 5 studies the Constant Function Market Maker, a class of automated market-maker widely used to provide liquidity in existing decentralized exchanges [55, 56, 157]. These systems operate in a fundamentally different computational model; they execute transactions sequentially, with minimal computational costs. In some respects they mirror the traditional exchange design with a central limit order book [2], but with a smooth distribution of limit orders, instead of a discrete set. If they were not implemented on replicated state machine infrastructure with inherent latency, they would execute transactions immediately. CFMMs execute trades at predictable prices, defined precisely by their trading curves.

Our topic at hand is the question of how these two market innovations can be combined. Batch auctions do not explicitly contain market-makers, and CFMMs have become a widely used strategy for market-making in decentralized exchanges. In the context of batch auctions, CFMMs can form a simple way to parametrize and express a market-making strategy in the batch auction. Yet these two different mechanisms do not obviously fit together immediately, given their different economic mechanisms and computational paradigms. We find that there are several design choices to make when integrating CFMMs with batch auctions, each of which can incur its own tradeoffs and economic effects.

6.1.1 Preliminaries

Constant Function Market Makers

Recall from Definition 5.2.1 that a Constant Function Market Maker (CFMM) is an automated market-making strategy parametrized by a trading function f. As in chapter 5, trading functions f satisfy Assumption 5.2.2. Additionally, for clarity of exposition in this chapter, we make the additional assumption that f is differentiable (except in §6.5). Recall that the gradient of f defines the spot exchange rate (Definition 5.2.3) for a trade of infinitesimal size. [3]

[2] Where the exchange maintains lists of offers to buy and sell an asset, and matches a new buy offer against the lowest price sell (if the limit prices can be matched) and vice versa.

[3] Strict quasi-concavity could be relaxed to quasi-concavity, and differentiability to continuity. This means replacing gradients with subgradients, as in Definition 5.2.3. When multiple points have the same spot price, and our theorem statements must pick one, any can be chosen in an arbitrary but deterministic fashion.

Arrow Debreu Exchange Markets

An Arrow Debreu market [78] is used to model a pure exchange market[4], i.e., a market where a set of assets $\mathfrak{A}$ are traded without a designated numeraire or "money". A ssets are fungible, divisible and freely disposable. Agents have quasi-concave and non-satiating utilities for the bundle of assets they consume and have an initial endowment of assets. All trades happen at the same prices implied by a valuation vector $\{p_\mathcal{A} > 0\}_{\mathcal{A}\in\mathfrak{A}}$, and asset amounts remain conserved.

Batch Exchanges

In a batch exchange, traders submit trade orders to an exchange operator, which gathers these orders into batches (typically with some temporal frequency, e.g. one batch per second) and then clears batches of orders subject to the constraints of the orders. Chapter 3 (as well as the batch exchange of [302]) models itself internally as an instance of an Arrow-Debreu exchange market (details in §A.1.1).

The following are examples of order types that a batch exchange may support.

Definition 6.1.1 (Limit Sell Offer)**.** *A Sell Offer ($\mathcal{A}$, $\mathcal{B}$, k, r) is request to sell $k \geq 0$ units of good $\mathcal{A}$, for as many units of asset $\mathcal{B}$ as possible, subject to receiving a minimum "limit" price of at least r units of $\mathcal{B}$ per $\mathcal{A}$.*

Limit sell offers correspond to agents in an Arrow-Debreu exchange market with linear utility [5] $u(x_\mathcal{A}, x_\mathcal{B}) = rx_\mathcal{A} + \mathcal{B}$ (Observation A.1.3).

Definition 6.1.2 (Market Sell Order)**.** *Limit sell orders with limit price of* 0 *are market sell orders.*

Definition 6.1.3 (Limit Buy Order)**.** *A limit buy order is an order to purchase up to k units of asset A by selling as few units of asset $\mathcal{B}$ as possible, subject to a maximum price of r $\mathcal{B}$ per unit $\mathcal{A}$.*

This corresponds to a utility function $u(x) = rx_\mathcal{A} + \min(k, x_\mathcal{B})$.

6.1.2 System Model

We model the interaction between a batch exchange, CFMMs, and users as follows.

1. Limit sell orders[6] can be submitted or removed any time before a cutoff for each next batch.

2. A CFMM participating in the batch exchange must submit its trading function and current reserves to the exchange before a cutoff time for each next batch.

[4]In its original form [78], the Arrow Debreu markets have two types of agents – consumers and producers – we are interested in a market with only consumers.

[5]The underlying utility of the trader placing the order may generally be different from the utility of the limit order. We abstract out from this detail and all strategic considerations – the mechanism is only interested in the limit order's utility.

[6]Limit buy orders lead to PPAD-hardness of equilibrium computation [118] (details in §A.8), and as such, we do not support limit buy offers.

3. Between consecutive batches, the CFMMs may also be available for their standalone operation, but must be unchanged after submitting their reserve levels and trading function to the batch exchange until the batch is executed.

We do not model any fees that the batch exchange operator and the CFMM may charge. We believe that the axioms and properties of market solutions we study in this work are important also in the presence of trading fees.

6.1.3 Our Results

To our knowledge, this is the first paper to do an axiomatic and computational study of the important problem of augmenting batch exchanges with CFMMs. We design a batch exchange which supports limit sell orders (Definition 6.1.1) as traders and CFMMs as market makers.

Axioms

We give here (details are in §6.2) a minimal set of axioms for batch exchanges incorporating CFMMs. We require that a batch exchange neither burn nor mint any asset – *asset amounts must be conserved* (Axiom 6.2.1). We also impose the core fairness property of batch auctions that there exist asset valuations in equilibrium, and no market participant receives a better price than that implied by these valuations (Axiom 6.2.2). Together, Axioms 6.2.1 and 6.2.2 imply that all trades in a batch happen at the same *batch prices* (Observation 6.2.3). Further, as is standard in trading systems, we want that all limit orders receive a trade which is a "best response" to the batch prices (Axiom 6.2.4). This axiom ensures that a limit sell order fully executes when the batch price exceeds the limit price and is not executed at all when the batch price is less than the limit price. The axiom pertaining to CFMMs is their basic design principle that their trading function should not decrease due to a trade (Axiom 6.2.5).

Axioms 6.2.1, 6.2.2, and 6.2.4 are simply an articulation of the core properties of batch exchanges which we believe are well-accepted rules for designing exchange markets. Axiom 6.2.5 is a basic guarantee required by all CFMM deployments. These axioms are *minimal* and do not impose a particular solution. They allow a rich set of solution concepts with different economically useful properties, which is this paper's core subject of study. We briefly discuss some solution concepts which carefully relax some axioms to expand the design space in §6.4.

We can now define a *market equilibrium* for batches with limit sell orders and CFMMs.

Definition 6.1.4 (Market Equilibrium)**.** *For a batch trading in a set of assets $\mathfrak{A}$, and market participants $i \in \mathcal{I}$ with initial endowments $\{x_{i,\mathcal{A}} \geq 0\}_{i\in\mathcal{I},\mathcal{A}\in\mathfrak{A}}$, a market equilibrium consists of a set of valuations $\{p_\mathcal{A} > 0\}_{\mathcal{A}\in\mathfrak{A}}$ and allocations $\{z_{i,\mathcal{A}} \geq 0\}_{i\in\mathcal{I},\mathcal{A}\in\mathfrak{A}}$ which satisfy Axioms 6.2.1, 6.2.2, 6.2.4,* *and 6.2.5 (asset conservation, uniform batch valuations, limit orders get a best-response trade, and the CFMM trading functions are non-decreasing).*

We leverage existing results from the literature to obtain sufficient conditions for the existence of market equilibria. Multiple market equilibria may exist for any batch instance per Definition 6.1.4.

Desirable Properties of Batch Exchanges and Market Equilibria

We now define some desirable properties of market equilibria. These properties then guide us in designing algorithms for finding market equilibria which are economically more useful than some other market equilibria.

1. The first desirable property is *Pareto optimality* for the limit sell orders (Definition 6.3.1). An outcome of a mechanism is Pareto optimal if no agent can be made strictly better-off without making someone else strictly worse-off [172].

2. Another desirable property is *price coherence* (PC) of a group of CFMMs post-batch (Definition 6.3.4). PC is a necessary and sufficient condition for the participating CFMMs to be in a *cyclic arbitrage-free* state (Definition 6.3.3).

   A weaker condition than PC is *preservation of price coherence* (PPC), under which a group of participating CFMMs must be price-coherent post-batch if they were price-coherent pre-batch.

3. We also consider the property *joint price discovery* (JPD) (Definition 6.3.10) – a property strictly stronger than PC. JPD eradicates a form of arbitrage called *parallel running* (Definition 6.3.9).

   Observation 6.1.5. *PPC is strictly weaker than PC, which is strictly weaker than JPD.*

4. Another desirable property is *locally computable response* (LCR) (Definition 6.3.12) for the CFMMs. That is, the trade made by a CFMM in equilibrium is a deterministic function of only its trading function, its pre-batch state, and the batch asset valuations. If the exchange implements LCR for CFMMs, it provides predictability to the liquidity providers and can help them in doing a better risk-profit analysis and making a more informed decision about committing to a market-making strategy.

We also discuss a property called *path independence* (Definition 6.3.15) in § 6.3.5 motivated from the standalone operation of CFMMs for batches with a single limit order.

Achievability of Desirable Properties of Batch Exchanges

All results here are under the setup that Axioms 1-4 are required to be satisfied, that is the batch exchange computes a market equilibrium as a solution. We relax this only in § 6.4 where post-processing is allowed for the CFMMs after the market equilibrium. We start with two impossibility results regarding the achievability of certain desirable properties simultaneously.

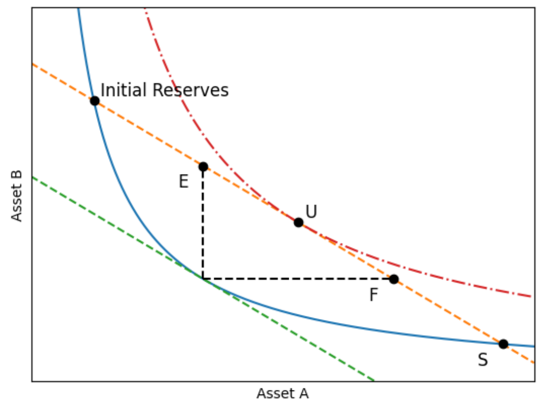


Figure 6.1: Examples of LCR CFMM trading rules. The axes are the CFMM's reserves. The blue curve is a level curve of the CFMM trading function on which the initial state lies. The slope of the tangent to the level curve denotes the CFMM's spot price. The slope of the dotted orange line is the negative of the batch price for this example. Here, the green and the orange lines have the same slopes.
Trading Rules S and U are per Definitions 6.1.10 and 6.1.8. Trading Rules E and F are inspired by the rebalancing strategy of Milionis et. al [244] and are defined later in §6.4. The line segment between points E and F corresponds to the class of Strict-Surplus Trading Rules (Definition 6.4.5).

Theorem 6.1.6. *A batch exchange cannot simultaneously guarantee Pareto optimality and preservation of price coherence (PPC).*

Theorem 6.1.7. *A batch exchange cannot simultaneously guarantee a locally computable response (LCR) for CFMMs and Pareto optimality.*

Recall that Pareto optimality is defined for the utilities of the limit orders, and PPC and LCR protect the interests of the CFMMs. These results shed light on an inherent tension between the interests of the CFMM and those of the limit orders. They also illustrate that the problem we study in this paper is non-trivial and nuanced.

We now turn to designing algorithms that find market equilibria with some of the desirable properties given above. For this, we adopt a viewpoint from the perspective of the CFMMs. Any deterministic algorithm for equilibrium computation can be described as a *trading rule* for a CFMM, a function which specifies the CFMM's allocation, given all information of the batch instance.

Consider the following natural CFMM trading rule.

Definition 6.1.8 (Trading Rule U)**.** *With its trading function as a pseudo-utility function, give the CFMM a pseudo-utility-maximizing bundle of assets subject to the asset valuations.*

Under Assumption 5.2.2 on CFMM trading functions, the allocation under Trading Rule U is unique.

$$F_U(x, f, p) = \sup_{z \in \{\hat{z} \mid p\cdot(\hat{z}-x)=0\}} f(z).$$

An illustration of Trading Rule U is in Figure 6.1. Notice that Trading Rule U satisfies LCR. That is, the CFMM's allocation in equilibrium interacts with the rest of the batch only via the asset valuations p. When the CFMM demand under Trading Rule U satisfies an additional condition – Weak Gross Substitutability (WGS) [7] [8] – Trading Rule U in the batch exchange can be implemented by well-known algorithms for computing equilibria in Arrow Debreu markets, such as the Tâtonnement-based algorithm described in Codenotti et. al [127], or our convex program of §6.5. Apart from computational tractability, the simple Trading Rule U also has other desirable properties.

Theorem 6.1.9. *A batch exchange has JPD if and only if it implements Trading Rule U for all CFMMs.*

By Observation 6.1.5, batch exchanges implementing Trading Rule U for all CFMMs also achieve PC. This implies that adopting Trading Rule U protects CFMMs from both cyclic arbitrage and parallel-running-based arbitrage. Trading Rule U has a natural interpretation that it treats CFMMs as utility-maximizing agents which may be normatively significant in many scenarios.

We also study Trading Rule S, which is applicable only for batches with 2-asset CFMMs[9].

Definition 6.1.10 (Trading Rule S)**.** *Maximize the CFMM's valuation-weighted trade size subject to the non-decreasing trading function constraint. Under Assumption 5.2.2, the allocation is unique.*

$$F_S(x, f, p) = \sup_{z \in \{\hat{z} \;\mid\; p\cdot(\hat{z}-x)=0;\; f(\hat{z})\geq f(x)\}} \|p \cdot (z - x)\|_1.$$

An illustration of Trading Rule S is in Figure 6.1. Informally, it corresponds to trading "all the way" up to the point where the trading function non-decreasing constraint is tight. It always ends up on the same level curve of the trading function as the initial state. For the extreme sparse case of a single CFMM and a single limit order, Trading Rule S mimics a standalone CFMM, and therefore, in this case, a limit order based-trader does not strictly prefer trading "outside the batch" with the CFMM. Notice that Trading Rule S also satisfies LCR, i.e., it gives a CFMM's "demand" as a response to asset valuations. All CFMM trading functions lead to a WGS demand under Trading Rule S, and as such, the equilibrium computation problem is computationally tractable (for example, once again, using Tâtonnement [127] or the convex program of §6.5). This simple and intuitive solution concept has some more desirable properties in special cases of batch instances as we show here.

Although per Theorem 6.1.7, when guaranteeing LCR for all CFMMs, the batch exchange cannot

[7]A demand function satisfies WGS if for all asset valuations $\{p_a\}_{a\in\mathcal{A}}$, on decreasing the valuation p_{A^*} of an asset A^* while keeping all other valuations $\{p_a\}_{a\in\mathcal{A}\setminus A^*}$ constant, the demand of all assets other than A^* does not decrease. WGS of all agents is a sufficient condition for the existence of market equilibria [215] in Arrow-Debreu exchange markets.

[8]Although not all CFMM trading functions correspond to WGS demand functions under Trading Rule U, we show that many natural classes of trading functions do. On the flip side, some seemingly-natural trading functions, for example, that of *Curve* [157] do not.

[9]For clarification, here we consider the cases where a CFMM trades in only two assets, but each CFMM may be trading in an arbitrary pair of assets, and the entire batch trades in an arbitrary number of assets.

guarantee Pareto optimal equilibria, it can nonetheless do so by using Trading Rule S for the special case of batches trading in only two assets when the CFMMs are price coherent pre-batch.

Theorem 6.1.11. *If the batch exchange trades in only two assets, and if the CFMMs in a batch are in a state of price coherence pre-batch, the equilibria obtained by algorithms implementing Trading Rule S for all CFMMs are Pareto optimal.*

Trading Rule S is the only LCR CFMM trading rule with this property.

Theorem 6.1.11, in conjunction with the impossibility result of Theorem 6.1.7, shows how multi-asset batch exchanges pose much more analytical challenges than two-asset batch exchanges.

Trading Rule S, in general, does not satisfy PPC. However, this negative result is bypassed in batches with only "Concentrated Liquidity Constant Product" (CLCP) CFMMs (Definition 6.3.8), which is a class including the constant-sum and constant-product CFMMs.

Theorem 6.1.12. *CLCP is the unique class of CFMMs such that if all CFMMs in a batch belong to this class, then the market equilibria attained by batch exchanges implementing Trading Rule S for all CFMMs satisfy the preservation of price coherence.*

This result uncovers a very interesting and useful fact about the class of CFMMs that are used extensively in practice and theory. While CLCP trading functions are hailed for their computational simplicity and universality [56], we establish the surprising fact here that they are also well-suited for integration with batch exchanges while using a very natural trading rule.

Results on Equilibrium Computation

Although tâtonnement-based algorithms can find market equilibria whenever agent demand functions are WGS, convex programs for computing market equilibria have led to an improved structural understanding of markets (such as the much-celebrated result of Eisenberg and Gale [158]). Therefore, we develop a convex program (in §6.5) for computing market equilibria for markets with limit sell orders and 2-asset CFMMs that have a WGS demand under any given LCR CFMM trading rule. This program may be of independent interest – it makes progress on the open question of designing convex programs for nonlinear utility functions in Arrow Debreu exchange markets. Our program handles the case where agent utility functions are arbitrary quasi-concave functions of two assets, subject to the constraint that each agent's demand function is WGS.

Theorem 6.1.13 (Informal)**.** *The program with objective function COP and constraints C1, C2, and C3 is convex, and its solutions correspond to market equilibria (if one exists) when each agent trades in only two assets and their demand functions are WGS.*

Our convex program is inspired by that of Devanur et. al [146] for the case of linear utility functions. The proof is quite technical, but the intuition is easy to state – we develop a viewpoint from which CFMMs appear as an uncountable collection of infinitesimal limit sell offers. Unlike the analyses in [146] based on Lagrange multipliers, we have to go back to earlier techniques and directly apply Kakutani's fixed point theorem.

When the density of this infinite collection of limit offers is a rational linear function, we prove that our convex program has rational solutions. This includes many classes of commonly used CFMMs, including the constant product CFMM when using Trading Rule U. On the flip side, we show that some CFMMs, for example those based on the Logarithmic Market Scoring Rule (LMSR) [191], can force batches only to admit irrational equilibria when using Trading Rule U.

Solution Concepts Beyond Market Equilibria

A natural question is whether we can attain more of the desirable properties of §6.1.3 if we allow the CFMMs to do a post-processing step after their trade in the batch. The answer is yes but with a caveat. If the CFMMs have the option to alter their state post-batch, then they can always attain a state of price coherence, but this will have to violate the axiom of uniform valuations (Axiom 6.2.2). Further, we give a class of LCR trading rules – Strict-Surplus Trading Rules (Definition 6.4.5) – on 2-asset CFMMs with the following property:

In any market equilibrium where all CFMMs' trades are per a Strict-Surplus Trading Rule, each CFMM has a surplus-capturing post-processing step, upon which they attain a state of JPD (Theorem 6.4.6).

This result opens up a new dimension to the solution space for our problem. Such multiple-step solutions concepts may provide a viable design in practice, but their economic implications must be further scrutinised in future research. More details on these results are in §6.4.

6.1.4 Related Work

In his seminal work, Kyle [216] studies a model of a two-asset (a *good* and a *money*) batch auction, where traders submit market orders to buy or sell the good. The market maker a priori declares a pricing rule that maps the excess demand for the good to a price. In contrast, we study a model where traders submit limit orders, and the market makers use a CFMM-based approach. A CFMM trading function, under LCR, can be considered a map from its trade size to a price for a given initial state. However, a CFMM is 'automated' in the sense that its pricing rule accounts for the liquidity provider's 'position' in the good, which is captured in the initial state. Moreover, unlike [216], we consider multi-asset batches and the case of potentially many market makers with their own CFMM trading functions.

Most closely related to our work in the blockchain space is the work of the company CoWSwap (Coincidence of Wants– Swap) (formerly known as Gnosis); their implementation details are provided in [302]. Same as us, they study batches that incorporate CFMMs and trade in multiple assets. They take an optimization approach with various objectives, such as maximizing trade volume or maximizing trader utility. Their solutions are based on mixed integer programs and do not have runtime guarantees. In contrast, we take an axiomatic approach and propose polynomial time solutions for finding market equilibria with certain desirable properties.

The decentralized exchange Penumbra [17] considers batches of two assets only. It aggregates *market* orders in a batch and executes the excess demand with a CFMM per its constant function trading rule, subject to a maximum slippage tolerance.

Automated market-making strategies have been studied both in a cryptocurrency context [65] and in traditional exchanges [182, 180, 216, 258]. CFMMs form a subclass of automated market-making. There has been extensive study of how the design of a CFMM trading function interacts with the economic incentives of those who invest in it [162, 111, 166, 165, 112], research on axiomatizing meaningful CFMM trading functions [281, 171, 260], and "optimality" amongst trading functions (as in chapter 5, as well as Milionis et. al [243]). However, this direction of work is orthogonal to the subject of study of this chapter. We focus only on the contract the CFMM-based trader enters into with the batch exchange operator. Axiom 6.2.5 captures a basic guarantee of this contract.

Budish et. al [103, 102] and Aquilina et. al [73] study the economic performance of batch auctions between pairs of assets. The well-studied model of [78] forms the basis for multi-asset, pure exchange batch trading implementations [201, 302]. There are many classes of algorithms for (approximately or exactly) computing equilibria in Arrow-Debreu exchange markets, including iterative methods (or Tâtonnement) [131, 127, 128, 90, 176], convex programs for the case of linear utilities [146, 199, 250, 137], convex program for Cobb-Douglas utilities [142], and combinatorial algorithms [200, 147, 154, 177].

Most closely related to our work in this line is the convex program of Devanur et. al [146], which also gives concise proof of the existence and rationality of equilibria. We generalize their program beyond linear utilities and include 2-asset WGS utility functions. The convex program of [250] (re-discovered in [199]—the original was in Russian) also goes beyond linear utilities but in an incomparable manner from ours. Their program can handle constant elasticity of substitution (CES)[10] utilities for $\rho > 0$ and Cobb-Douglas utilities on any number of assets but not the entire class of WGS utility functions. A precise characterization of the class of utility functions that their program can handle is unknown. This question was posed as an open problem alongside several positive and negative examples in [199]. On the other hand, when each agent has utility for only two assets, our program handles all WGS utility functions in the class.

[10]A CES utility has the form $u(x) = (\sum_{A\in\mathcal{A}}(w_A x_A)^\rho)^{1/\rho}$ for nonnegative constants $\{w_A\}_{A\in\mathcal{A}}$. In the limit $\rho \to 0^+$, we get the Cobb-Douglas utilities of the form $u(x) = \prod_{A\in\mathcal{A}} x_A^{w_A}$ for $\sum_{A\in\mathcal{A}} w_A = 1$. The Cobb-Douglas function is widely used as the trading function of many common CFMMs [241, 55].

6.2 Axioms of Batch Exchanges with CFMMs

In this section, we give a set of axioms that specify minimal guarantees that a market solution must provide to the participants in a batch exchange.

6.2.1 Asset Conservation

Trading systems must not create or destroy any asset. This is equivalent to the "market-clearing" property of Arrow Debreu exchange market equilibria. Formally:

Axiom 6.2.1 (Asset Conservation)**.** *Let exchange participants $i \in \mathcal{I}$ have endowments $\{x_{i,\mathcal{A}}\}_{i\in\mathcal{I};\mathcal{A}\in\mathfrak{A}}$ and receive bundles $\{z_{i,\mathcal{A}}\}_{i\in\mathcal{I};\mathcal{A}\in\mathfrak{A}}$. For each asset $\mathcal{A}$, we must have $\sum_{i\in\mathcal{I}} x_{i,\mathcal{A}} = \sum_{i\in\mathcal{I}} z_{i,\mathcal{A}}$.*

6.2.2 Uniform Valuations

The core fairness property of a batch trading scheme is that all orders in a batch trade at the same prices, and no trader gets an unfair advantage. For this, the batch exchange must compute valuations for all assets in a batch and ensure that no trader can get an allocation of a greater value than the value of their endowment per these valuations. Formally:

Axiom 6.2.2 (Uniform Valuations)**.** *An equilibrium of a batch trading scheme has a shared market valuation $\{p_{\mathcal{A}} > 0\}$ for each asset $\mathcal{A} \in \mathfrak{A}$. Let batch exchange participants $i \in \mathcal{I}$ have endowments $\{x_{i,\mathcal{A}}\}_{i\in\mathcal{I};\mathcal{A}\in\mathfrak{A}}$ and get allocated bundles $\{z_{i,\mathcal{A}}\}_{i\in\mathcal{I};\mathcal{A}\in\mathfrak{A}}$. For each participant $i \in \mathcal{I}$, it must be the case that $p \cdot z_i \leq p \cdot x_i$.*

Note that we do not require strict equality between $p\cdot z_i$ and $p\cdot x_i$ in the axiom; instead, we require that no market participant should get a price strictly better than the equilibrium asset valuations. However, in conjunction with asset conservation (Axiom 6.2.1), uniform valuation (Axiom 6.2.2) implies that this inequality needs to be strict in equilibrium.

Observation 6.2.3. *Axioms 6.2.1 and 6.2.2 imply that all trades in equilibrium in any asset pair $(A, B) \in \mathcal{A}^2$ happen at the same price, given by quotients of the asset valuations.*

Proof. Summing the inequalities of Axiom 6.2.2 over all participants, $\sum_{i\in\mathcal{I}} p \cdot z_i \leq p \cdot x_i$. By asset conservation, $\sum_{i\in\mathcal{I}} z_i = \sum_{i\in\mathcal{I}} x_i$. Since all $p_{\mathcal{A}} > 0$, all inequalities $p \cdot z_i \leq p \cdot x_i$ must be tight. □

One may envision a trading system allowing giving CFMMs a worse price than the limit orders to be able to execute some more limit orders. However, such a system will violate the asset conservation axiom and leave a surplus. This observation ensures that the $\binom{|\mathfrak{A}|}{2}$ exchange rates for each pair of assets are obtained from a vector of $|\mathfrak{A}|$ asset *valuations.*

The existence of the valuation vector, which facilitates marker clearing, is not guaranteed for arbitrarily batch instances. However, we show in §6.5 that we can leverage the literature on Arrow

Debreu markets to obtain the necessary conditions for the existence of market clearing valuations and corresponding allocations in our context.

6.2.3 Limit Sell Orders Make a Best Response

A basic guarantee of batch trading systems in the literature is that a limit sell order is executed in full when the batch price exceeds the limit price and is not executed when the batch price is less than the limit price. When the batch price equals the limit price, the limit order trades any amount between zero and its maximum amount.

Axiom 6.2.4 (Best response trade for limit orders)**.** *The allocation received by a limit sell order maximizes its linear utility function subject to the market asset valuations.*

6.2.4 CFMM Design Specification

The "constant function market maker" name may suggest that a CFMM shall trade from reserves x to reserves z only if $f(z) = f(x)$. One might consider encoding this strict equality condition as an axiom; however, real-world CFMM deployments only check the weaker condition that $f(z) \geq f(x)$ [32]. Keeping this flexibility allows us to have a much richer design space and is crucial to satisfy certain desirable properties of batch trading systems, as we shall see in the next section.

Axiom 6.2.5 (Non-decreasing CFMM trading function)**.** *A CFMMs trading function does not decrease due to a trade in the batch, i.e., for a trade from state x to z, we must have $f(z) \geq f(x)$.*

Recall Definition 6.1.4 that any market equilibrium satisfies all Axioms 6.2.1, 6.2.2, 6.2.4, and 6.2.5.

6.3 Desirable Properties of Market Equilibria

Market equilibria may not be unique, so we turn now to study desirable properties of market equilibria, which will then guide our design of market equilibrium computation algorithms.

6.3.1 Pareto Optimality

Given the asset valuations in market equilibrium, the utility that a limit sell order receives is fixed under Axiom 6.2.4. However, some market equilibria may have asset valuations that provide a worse utility to the limit orders than other admissible market equilibria. The Pareto optimality of market equilibria is defined as follows.

Definition 6.3.1 (Pareto Optimal Market Equilibria)**.** *For a batch instance, a market equilibrium E is Pareto optimal if there does not exist another market equilibrium E' which Pareto dominates E for the utility of the limit sell orders.*

Note that in a market with only limit sell orders, *all* market equilibria are Pareto optimal – this property is lost if CFMMs also participate. There is no natural notion of the utility of a CFMM in our model, and since CFMMs are expected to charge trading fees (we do not model the fees in this paper), we define the Pareto optimality of market equilibrium for the utility of the limit orders only.

We illustrate with an example that multiple market equilibria may exist even for simple batch instances, and many of those may not be Pareto optimal.

Example 6.3.2. *Consider an instance of a batch exchange trading in assets A and B, with a CFMM $f(x) = x_A x_B$, in the state $x_A = 1, x_B = 4$, and a limit order $l = (A, B, 1, 1)$ i.e., for selling up to 1 unit of A for B with a minimum price of 1.*

The set of market equilibria is given by the asset valuations $(p_A = r, p_B = 1)$ for $r \in [1, 2]$. The corresponding bundle that the limit order receives is $(z^l_A = 0, z^l_B = r)$, and that the CFMM receives is $(z^c_A = 2, z^c_B = 4 - r)$. The unique Pareto optimal market equilibrium corresponds to $r = 2$.

We discussed Pareto optimality to safeguard the interests of the limit order traders. We now give desirable properties aimed at safeguarding the interests of the CFMM liquidity providers.

6.3.2 Price Coherence and Mitigating Cyclic Arbitrage

For a CFMM trading standalone (not in a batch exchange), its state changes only in response to trade requests and not directly in response to the prices of assets in the external market. When the external market prices change, the CFMM's spot price is now "stale." An arbitrageur can make a risk-free profit by buying from the CFMM some units of the asset whose relative price has increased on the external market and selling it there at the new (higher) price. The size of the trade to maximize the arbitrager's profit can be computed by solving a convex program [69]. This optimal arbitrage trade leaves the CFMM's spot price aligned with the external market's price [65].

An arbitrage opportunity also arises when a group of CFMMs are mispriced with respect to each other. Towards this, we define "cyclic arbitrage", which motivates important design considerations and properties we desire from market equilibria in batch exchanges.

Definition 6.3.3 (Cyclic Arbitrage of a Group of CFMMs)**.** *A group of CFMMs $\mathcal{C}$ is in a state of cyclic arbitrage if it is possible to obtain a non-zero amount of any asset for free by trading with the group $\mathcal{C}$ simultaneously.*

A state of cyclic arbitrage can arise, for example, when CFMMs operate in a batch exchange, and the system's design does not eradicate this possibility. We define a related property on the spot prices of CFMMs, which provides a handle on the analysis of cyclic arbitrage.

Definition 6.3.4 (Price Coherence of a Group of CFMMs)**.** *A group of CFMMs $\mathcal{C}$ has price coherence if there exists a set of asset valuations $\{q_a > 0\}_{a \in \mathcal{A}}$ such that for each asset pair $(A, B) \in \mathcal{A}^2$,* *every CFMM $c \in \mathcal{C}$ that trades in asset pair (A, B) has a spot price $\frac{q_A}{q_B}$ units of B per unit of A.*[11]

For a group of CFMMs, construct a graph with the CFMMs as nodes and an undirected edge between two nodes if they trade at least one common asset. If a set of CFMMs in a cycle of this graph (if one exists) does not have price coherence, then they are in a state with cyclic arbitrage – a trader can make a risk-free profit by carefully trading with this cycle of CFMMs. More precisely:

Proposition 6.3.5. *A group of CFMM are in a state of cyclic arbitrage if and only if they are not in a state of price coherence.*

Proof. Follows from the "no-arbitrage condition" of Angeris et. al [68]. □

The price coherence of a group of CFMMs motivates a related property for market equilibria.

Definition 6.3.6 (Price Coherence of Market Equilibria (PC))**.** *A batch exchange market equilibrium satisfies PC if the group of participating CFMMs have price coherence post-batch.*

It is important for the liquidity providers of the CFMMs that cyclic arbitrage opportunities are not created by trading in the batch. We also define a weaker condition than price coherence for equilibrium computation algorithms, which may be more relevant in some circumstances.

Definition 6.3.7 (Preservation of Price Coherence (PPC))**.** *A batch exchange satisfies PPC if, for any batch instance with a group of CFMMs in a state of price coherence pre-batch, the equilibria computed by the exchange satisfy PC.*

We now restate and prove Theorem 6.1.6.

Theorem (6.1.6 restatement)**.** *A batch exchange cannot simultaneously guarantee Pareto optimality and preservation of price coherence (PPC).*

Proof. Consider a batch instance trading in assets $\mathcal{A} = \{A, B\}$. There are two CFMMs:

C_1 with trading function $f_1(x) = x_A x_B$ and pre-batch state $(x_A = 1, x_B = 2)$ with spot price 2, and

C_2 with trading function $f_2(x) = x_A^2 x_B$ and pre-batch state $(x_A = 1, x_B = 1)$ with spot price 2.

There is a limit sell order $(A, B, 1, 1)$, i.e., to sell up to 1 unit of A for B for a price of at least 1.

The set of market equilibria is given by asset valuations $(p_A = r, p_B = 1)$ for $r \in [1, (8 + \sqrt{10})/9]$ and corresponding allocations.

The unique Pareto optimal market equilibrium corresponds to $r = (8 + \sqrt{10})/9 \approx 1.24$. In this equilibrium, the post-batch spot price on CFMM C_1 is ≈ 0.769 and that on CFMM C_2 is ≈ 0.748. □

[11]When the CFMM trading function is not differentiable, we require that $\frac{q_A}{q_B}$ be in the set of subgradients of the level curve of the trading function, and all results regarding PC will continue to hold.

Although not possible to be guaranteed with Pareto optimality, PC can nonetheless be guaranteed by natural CFMM trading rules in batch exchanges. An example is Trading Rule U (recall Definition 6.1.8) – the result follows as a corollary of Theorem 6.1.9.

Recall that PPC is a weaker condition than PC. Although Trading Rule S (recall Definition 6.1.10) does not satisfy PPC in general, it does so when all CFMMs in the batch belong to a particular class of CFMMs, defined below.

Definition 6.3.8 (Concentrated Liquidity Constant Product (CLCP) Trading Functions)**.** *A trading function in the CLCP class is $f(x) = (x_A + \hat{x}_A)(x_B + \hat{x}_B)$ for constants $\hat{x}_A > 0, \hat{x}_B > 0$.*

This class of CFMMs allocates liquidity in a single interval of prices and implements the constant-product trading function in that interval. This price interval is $(\frac{\hat{x}_B^2}{K}, \frac{K}{\hat{x}_A^2})$ for constant K denoting the initial value of the CFMM trading function. This class includes, as special cases, the constant-product CFMM [55] (where the liquidity is spread to all prices, 0 to ∞, by setting $\hat{x}_A = \hat{x}_B = 0$) and the constant-sum CFMM (where the liquidity is concentrated at a single price point). Combinations of CLCP trading functions form the basis of the widely successful decentralized exchange protocol Uniswap V3 [56]. CLCP trading functions have the following interesting property.

Theorem (6.1.12 restatement)**.** *CLCP is the unique class of CFMMs such that if all CFMMs in a batch belong to this class, then the market equilibria attained by batch exchanges implementing Trading Rule S for all CFMMs satisfy the preservation of price coherence.*

Proof. Denote the asset valuations which give the initial spot prices of the CFMMs by $\{q_a\}_{a\in\mathcal{A}}$. These exist by pre-batch price coherence. Denote the batch equilibrium asset valuations by $\{p_a\}_{a\in\mathcal{A}}$ and the asset valuations which give the post-batch spot prices of the CFMMs by $\{\hat{q}_a\}_{a\in\mathcal{A}}$. These exist by PPC. Since each CFMM here trades in only two assets (Trading Rule S is defined only for 2-asset CFMMs), its initial spot price is $q_{AB} = q_A/q_B$ for assets A and B that it trades. Similarly, denote the batch price in A and B by p_{AB}.

For a given CFMM trading function, recall that Trading Rule S gives a map F_S (Definition 6.1.10) from the initial state and the batch asset valuations to the final state. Observe that this map depends on $\{p_a\}_{a\in\mathcal{A}}$ only via the batch price p_{AB}. Under the strict quasi-concavity of the CFMM trading function, if the CFMM trading function value is invariant, there is an injective map from the CFMM state to its spot price. Since the CFMM trading function value is invariant under Trading Rule S, we can represent Trading Rule S as a map from the initial spot price and batch price to the final spot price. Denote this map for trading function f by $g_f(q_{AB}, p_{AB}) \to \hat{q}_{AB}$.

Here, we give properties that the map $g_f(\cdot,\cdot)$ must satisfy.

1. *Reflexive.* That is, $g_f(\rho, \rho) = \rho$ for all $\rho \in [0, \infty)$. This is required since the CFMM makes no trade when the batch price equals the spot price, and hence the post-batch spot price should equal the initial spot price.

2. *Involution on the Spot Price.* That is, $g_f(q,p) = \hat{q} \implies g_f(\hat{q},p) = q$ for all $q, p, \hat{q} \in [0,\infty)$. This is because, under Trading Rule S, the CFMM returns to the initial state after two consecutive batches if the batch price is the same in both batches.

3. *PPC.* For a group of CFMMs, construct a graph with the CFMMs as nodes and an undirected edge between two nodes if they trade at least one common asset. For any cycle $\mathbb{C}$ of CFMMs in this graph, see that $\prod_{i\in\mathbb{C}} p_{A_iB_i} = 1$ by arbitrage-freeness of batch valuations (here CFMM i trades in asset A_i and B_i, $A_{i+1} = B_i$, and $B_{|\mathbb{C}|} = A_1$). Also $\prod_{i\in\mathbb{C}} q_{A_iB_i} = 1$ by pre-batch PC. PPC requires that $\prod_{i\in\mathbb{C}} g_{f_i}(q_{A_iB_i}, p_{A_iB_i}) = 1$.

From basic functional analysis, we obtain that the only differentiable functions that satisfy these conditions are $g_f(q,p) = q$ and $g_f(q,p) = \frac{p^2}{q}$.

The case of $g_f(q,p) = q$ corresponds to the constant sum CFMM whose spot price is invariant to the CFMM state. Observe that this CFMM is in the CLCP class (Definition 6.3.8) and corresponds the case where the liquidity is concentrated at one price.

We now solve for the trading function corresponding to $g_f(q,p) = \frac{p^2}{q}$.

Let a CFMM trade in assets A and B, and its reserves in each asset be x_A and x_B. When CFMM trading function value $f(x)$ is invariant, it gives an injective map from A in the reserves to B in the reserves. Denote this map by $B_f(x_A)$ for a fixed level curve of the CFMM trading function f. In this notation, the spot price of the CFMM is $\frac{dB_f}{dx_A}$.

For two points on a level curve, (x_{A0}, x_{B0}) and (x_{A1}, x_{B1}), the condition $g_f(q,p) = \frac{p^2}{q}$ translates to

$$\left.\frac{dB_f}{dx_A}\right|_{x_{A0}} \cdot \left.\frac{dB_f}{dx_A}\right|_{x_{A1}} = \left(\frac{x_{B1} - x_{B0}}{x_{A1} - x_{A0}}\right)^2 \tag{6.1}$$

Taking the point (x_{A0}, x_{B0}) as fixed, denoting $\left.\frac{dB_f}{dx_A}\right|_{x_{A0}}$ by constant c, and solving the differential equation yields the following indefinite integrals.

$$c\int \frac{dB_f}{(x_{B1} - x_{B0})^2} = \int \frac{dx_A}{(x_{A1} - x_{A0})^2} \tag{6.2}$$

Further solving gives

$$\frac{c}{(x_B - x_{B0})} = \frac{1}{(x_A - x_{A0})} + c_1 \tag{6.3}$$

where c_1 is the constant of integration. For $x_B \neq x_{B0}$ and $x_A \neq x_{A0}$:

$$c(x_A - x_{A0}) = (x_B - x_{B0}) + c_1(x_A - x_{A0})(x_B - x_{B0}) \tag{6.4}$$

Upon rearrangement of the terms, this yields the CLCP trading function form. In case the CFMM runs out of an asset, the spot price for selling that asset is ∞ for a CLCP trading function – using this value in the PC equation ensures that the result continues to hold. □

6.3.3 Joint Price Discovery and Mitigating Parallel Running

We motivated PC and PPC, intending to ensure that the group of CFMMs participating in the batch exchange do not end up in a state of cyclic arbitrage. However, the problem of arbitrage in financial systems is not limited to the trades that can be made with a group of CFMMs.

Here we study another form of an arbitrage strategy.

Definition 6.3.9 (Parallel Running)**.** *For a batch instance, a parallel running arbitrage opportunity exists if for any real numbers $a, b > 0$ and any assets $A, B \in \mathcal{A}$, a trader can sell a units of asset A in exchange for b units of asset B in the batch and, can then obtain a units of asset A in exchange for $\hat{b} < b$ units of asset b by trading with the participating CFMMs post-batch.*

Parallel running is a similar concept to front running, which corresponds to making a trade based on advanced knowledge of future orders. By definition, precisely identifying parallel running opportunities requires knowledge of all the other batch participants and the equilibrium computation algorithm. However, estimating the chances and magnitudes of such opportunities with only partial information may be possible – we leave this investigation for future work. We show here that we can design equilibrium computation algorithms that eradicate all parallel running opportunities, even if the arbitrager has perfect information of the batch.

First, we describe a property of market equilibria.

Definition 6.3.10 (Joint Price Discovery (JPD))**.** *Let group $\mathcal{C}$ of CFMMs participate in a batch. A market equilibrium with asset valuations $\{p_a > 0\}_{a \in \mathcal{A}}$ satisfies JPD if for each asset pair $(A, B) \in \mathcal{A}^2$, every CFMM $c \in \mathcal{C}$ that trades in pair (A, B) has a post-batch spot price $\frac{p_A}{p_B}$ units of B per unit of A.*

Observe that JPD is a special case of PC. The following result establishes its importance as a defence against parallel running.

Lemma 6.3.11. *Joint price discovery (JPD) makes parallel running impossible.*

Proof. See that in equilibrium, each order in the batch executes at the batch prices given by the ratios of asset valuations $\{p_a > 0\}_{a \in \mathcal{A}}$. By quasi-concavity of CFMM trading function f, for any trade, the price obtained is no better than the spot price. By JPD, the spot price is equal to the batch price, so parallel running is impossible. $\square$

We show in Appendix D.2 that for a reasonable regularity condition "split invariance" on equilibrium computation algorithms (in Definition D.2.2) and for "large" batch instances (as in Definition D.2.1), JPD is a *necessary* condition to eradicate parallel running in batch exchanges.

Beyond its desirable properties, JPD implies a natural CFMM trading rule. Towards this, we here restate and prove Theorem 6.1.9.

Theorem (6.1.9 restatement)**.** *A batch exchange has JPD if and only if it implements Trading Rule U for all CFMMs.*

Proof. The CFMM trading function f is strictly quasi-concave. Thus, to maximize f on a hyperplane (as in Trading Rule U) is to find the point where the gradient of f is normal to the hyperplane. This point is unique under Assumption 5.2.2. Since the gradient of f at a point is equal to the spot price at said point (up to rescaling), setting the spot prices equal to the batch prices corresponds to Trading Rule U. □

JPD has another advantage beyond mitigating parallel running. Previous work [155, 284] suggests that batch exchanges may emerge as major trading venues for the price discovery of assets. If this prediction is realized, then it will be vital for the CFMMs to have a spot price in lockstep with the price offered on the batch exchange. In a way, CFMMs shall achieve price discovery for "free" by trading in a batch exchange which implements JPD-ensuring market equilibria, which otherwise (in their standalone operation) comes at the cost of arbitrage (also termed "Loss-Vs-Rebalancing" [244]). A thorough analysis of the economic incentives of CFMM liquidity providers to participate in batch exchanges is the subject of future research.

We now study another property aimed at safeguarding the CFMMs.

6.3.4 Locally Computable Response for CFMMs

It is often important for agents to have some predictability in their trade in a batch, subject to the batch prices. While this is guaranteed for limit orders axiomatically (Axiom 6.2.4), it would also be good for liquidity providers who invest their capital in CFMMs for market making.

Definition 6.3.12 (Locally Computable Response (LCR) CFMM Trading Rule)**.** *A CFMM's trading rule in a batch exchange satisfies LCR if it is a map $F(x, f, p) \to z$, where $\{x_a\}_{a\in\mathcal{A}}$ is the pre-batch CFMM state, f is its trading function, and $\{p_a\}_{a\in\mathcal{A}}$ is a set of equilibrium asset valuations. The output $\{z_a\}_{a\in\mathcal{A}}$ is the post-batch CFMM state.*

Further, $F(x, f, p)$ must be invariant under rescaling of p.

LCR provides predictability to liquidity providers and makes participation more lucrative (apart from the fees they charge, which are not considered in our model). However, as with other desirable properties, we lose some design space if we impose LCR for CFMMs. For example, we have the impossibility result of Theorem 6.3.14, which uses Definition 6.3.13.

Definition 6.3.13. *The price-weighted trade volume of a limit sell order selling A for B is $p_A(x_A - z_A) + p_B(z_B - x_B)$ where x and z are their endowment and equilibrium allocation, respectively, and p are the equilibrium asset valuations.*

Theorem 6.3.14. *No batch exchange satisfying LCR for CFMMs can guarantee to find a market equilibrium that maximizes the sum of the price-weighted trade volumes of all limit sell orders.*

Proof. Consider a batch exchange with a single CFMM with trading function $f(x) = x_A x_B$ and pre-batch state $x_A = 1, x_B = 4$. We study two batch instances with this CFMM:

1. There is one limit sell order $(A, B, 3, 1)$ i.e., for selling up to 3 units of A for a minimum price of 1 B per A. All equilibria have the same (up to rescaling) asset valuations: $p_A = p_B = 1$.

   The "price-weighted trade volume of limit orders" is maximized by the equilibrium at which the CFMM buys 3 units of A from the limit order.

2. There is one limit sell order $(A, B, 3, 1)$ and another limit sell order $(B, A, 3, 1)$. All equilibria have the same (up to rescaling) asset valuations: $p_A = p_B = 1$.

   The "price-weighted trade volume of limit orders" is maximized by the equilibrium at which the CFMM makes no trades.

Since the equilibrium asset valuations are the same in the two instances, any LCR trading rule for the CFMM cannot distinguish between the two instances and cannot optimize for the price-weighted trade volume of limit orders. □

Walther [302] gives a mixed-integer program for the problem of finding a market equilibrium that maximizes the price-weighted trade volume of limit orders. Their algorithm has no runtime guarantees. Finding polynomial-time algorithms for this objective or showing that none exist is an interesting open problem. Also, recall the impossibility result of Theorem 6.1.7. We prove it here.

Theorem (6.1.7 Restatement)**.** *A batch exchange cannot simultaneously guarantee a locally computable response (LCR) for CFMMs and Pareto optimality.*

Proof. First consider the case of LCR CFMM trading rules which, for some starting state x, trading function f, and batch valuations p, demand an allocation z strictly "above the curve", i.e., $f(z) > f(x)$. Such a CFMM trading rule will clearly not lead to a Pareto optimal equilibrium. Consider a batch with one market sell order selling A for B, and a CFMM trading in a set of assets $\mathcal{A}$ which includes A and B. If, in equilibrium, the CFMM's allocation z satisfies $f(z) > f(x)$, then, by the strict quasiconcavity of f, there exists an alternate equilibrium under which the CFMM's allocation is z' where $z'_B < z_B$, $f(z') = f(x)$ (hence satisfying Axiom 6.2.5), and $\{z'_a\}_{a \in \mathcal{A} \setminus B} = \{z_a\}_{a \in \mathcal{A} \setminus B}$. In this equilibrium, the limit order gets strictly more units of assets B and is, therefore, a Pareto improvement.

Now, consider the case of LCR CFMM trading rules which, for all starting states x, trading functions f, and batch valuations p, demand an allocation z "on the curve", i.e., $f(z) = f(x)$. For two-asset CFMMs, this leaves only two possibilities (by strict quasi-concavity of trading function f), either the CFMM makes no trade (i.e., $z = x$) or the CFMM demands an allocation z such that $z = \sup_{z \in \{\hat{z} | p \cdot (\hat{z} - x) = 0;\ f(\hat{z}) \geq f(x)\}} \|p \cdot (z - x)\|_1$. Recall that this corresponds to Trading Rule S. We consider both these cases.

1. There exists a CFMM C in the batch with an LCR trading rule which, for some starting state x, trading function f, and batch valuations p (not equal to the CFMM spot price), demands an allocation z, where $z = x$ (that is, makes no trade).

   Say CFMM C trades in assets A and B. Consider a batch where no other group of CFMMs together trade in both A and B. There exists a batch instance with a market sell order for $a > 0$ units of asset A for B. The equilibrium under the above-stated LCR is sub-optimal for the limit order since it would have a higher utility on receiving any non-zero amount of B.

2. Consider a case where all CFMMs in the batch are two-asset CFMMs, and they follow the LCR Trading Rule S. For this case, we have the following example with two CFMMs and two limit sell orders.

   CFMM C_1 trades in assets A and B, uses the trading function $f(x) = x_A x_B$ and has a starting state $(x_A = 1, x_B = 4)$.

   CFMM C_2 trades in assets B and D, uses the trading function $f(x) = x_B x_D$ and has a starting state $(x_B = 1, x_D = 4)$.

   Limit order $L_1 = (A, D, 3, 1)$ i.e., to sell up to 3 units of A for D at a minimum price of 1 D per A.

   Limit order $L_2 = (A, B, 1, 1.5)$.

   On using Trading Rule S for both CFMMs, a market equilibrium E has asset valuations $p_A = p_B = p_D = 1$. CFMM C_1 buys 3 units of A, sells 3 units of B, and ends up in a state of $x_A = 4$ and $x_B = 1$. CFMM C_2 buys 3 units of B, sells 3 units of D, and ends up in a state of $x_B = 4$ and $x_D = 1$. Limit order L_1 trades in full and limit order L_2 does not trade.

   Another market equilibrium, $\hat{E}$, has asset valuations $p_A = 2; p_B = 1; p_D = 2$.

   CFMM C_1 buys 1 unit of A and sells 2 units of B. CFMM C_2 makes no trades. Limit order L_1 makes no trade and limit order L_2 trades in full (sells 1 unit of A for 2 units of B). The utility of L_1 is the same in E and $\hat{E}$, whereas that of L_2 is strictly better in $\hat{E}$. Therefore market equilibrium $\hat{E}$ is a Pareto improvement over E. □

These negative results illuminate the fundamental trade-offs that a batch exchnage must operate under. However we are able to obtain positive results in an important special case. We restate and prove Theorem 6.1.11.

Theorem (6.1.11 Restatement)**.** *If the batch exchange trades in only two assets, and if the CFMMs in a batch are in a state of price coherence pre-batch, the equilibria obtained by algorithms implementing Trading Rule S for all CFMMs are Pareto optimal.*

Trading Rule S is the only LCR CFMM trading rule with this property.

Proof. Let assets A and B be traded on the exchange. Denote a market equilibrium obtained by implementing Trading Rule S for all CFMMs by E. By pre-batch price coherence, CFMMs either all sell B or all sell A in equilibrium. The CFMMs sell B in E without the loss of generality.

An improvement in the utility of the limit orders selling A can be made only by making B cheaper relative to A. For higher utility, the limit orders selling A consume strictly more B than in equilibrium E. On the other hand, the limit orders selling B sell strictly fewer B. However, the CFMMs, all of whom are selling B in E, cannot sell any more of B at a lower price by the quasi-concavity of the trading function f and the fact that the CFMM is "on the curve" in Trading Rule S. Therefore such a utility improving market equilibrium is not possible.

An improvement in the utility of the limit orders selling B can be made only by making B costlier relative to A. However, in equilibrium E, some limit orders are buying B (since the CFMMs all sell B), and making B costlier will strictly decrease their utility.

We now show that Trading Rule S is the only LCR CFMM trading rule with this property, with a simple example. Consider a batch instance with a *market* sell order for selling asset A for B. If for some initial state x and trading function f, the batch exchange does not implement Trading Rule S, then it provides strictly fewer units of B to the limit order than under Trading Rule S and is therefore not Pareto optimal. □

6.3.5 Path Independence

We here discuss a property of the batch exchange (and not of market equilibria like the previous properties), defined when there is a single limit order-based trader. Blockchain systems often have restricted throughput capabilities, and it is important to design mechanisms which do not incentivize traders to submit multiple small orders instead of a single big order[12]. Towards this, we define the path-independence property, which is inspired by the standalone operation of CFMMs.

Definition 6.3.15 (Path Independence)**.** *An arbitrary group of CFMMs $\mathcal{C}$ participates in two consecutive batches and does not modify between the batches. Suppose a single trader exists and wants to sell some units of an asset $A \in \mathcal{A}$ in exchange for asset $B \in \mathcal{A}$ via market sell orders (that is, limit sell orders with limit price zero).*

In a path-independent batch exchange, the amount of asset B they receive on splitting the units of A they sell into the two batches is independent of the split.

This definition trivially extends to multiple batches and not just two. Standalone CFMMs satisfy path independence [65]. However, it is surprisingly difficult to satisfy in batch exchanges, except for a narrow case of Theorem 6.3.16. We restate and prove Theorem 6.3.16 here.

[12]It is a standard practice in finance that investors and traders with large orders submit their orders in smaller parts to receive a better average price since market liquidity at a time is limited [60, 113]. Another possible reason to do so is that the traders do not want to disclose their private information, which is reflected in their order size [178]. We do not aim to restrict this practice. We wish to restrict the incentives for breaking down orders even when the available liquidity in the market does not change.

Theorem 6.3.16. *Batch exchanges with Trading Rule S for CFMMs satisfy path independence when there is only one CFMM in the batch.*

Proof. The CFMM, under Trading Rule S, trades "on the curve." That is, $f(z) = f(x)$ where x and z are its pre-batch and post-batch states, respectively. For any amount of A that the market order trader sells, the amount of B in the CFMM is specified by the trading function preservation and therefore the amount of B that the trader receives is also only a function of the net amount of A sold by the trader. □

The path independence property, although defined in a restricted setup, provides a distinction between Trading Rules U and S. Even in the simple setup of a single CFMM and a single limit-order-based trader, batch exchanges implementing Trading Rule U do not satisfy path independence, as shown in the following example.

Example 6.3.17 (Trading Rule U does not satisfy path independence with one CFMM)**.** *Consider a batch instance with a single CFMM with trading function $f(x) = x_A x_B$ and pre-batch state $x_A = 1, x_B = 4$.*

If a trader submits a market sell order for 2 units of A, they receive 1.6 units of B. (See that the new CFMM state: $x_A = 3, x_B = 2.4$, has a spot price of 0.8 B per A, which matches the batch price offered).

If the same trader first submits a market sell order for 1 unit of A, they receive $4/3$ units of B for it. The new CFMM state is $x_A = 2, x_B = 8/3$. Now if they make another market sell order for 1 unit of A, they receive $2/3$ units of B for it, with the final CFMM state of $x_A = 3, x_B = 2$. The overall trade is 2 A for 2 B, which is strictly better than what they received on trading in full in one go.

Although simple and intuitive in the case of 2-asset batches, the path independence property is surprisingly difficult to satisfy. Even Trading Rule S does not satisfy it when more than one CFMMs are present, even if the CFMMs are price-coherent pre-batch, as in the following example.

Example 6.3.18. *CFMM C_1 trades in assets A and B, uses the trading function $f(x) = x_A x_B$ and has a starting state $(x_A = 10, x_B = 1)$.*

CFMM C_2 also trades in assets A and B, uses the trading function $f(x) = x_A + 10x_B$ and has a starting state $(x_A = 10, x_B = 0)$.

If a trader sells 1 B, they receive 10 A; CFMM C_1 makes no trade, and CFMM C_2 trades to state $(x_A = 0, x_B = 1)$. Then, if they sell another B, CFMM C_1 trades to final state $(x_A = 5, x_B = 2)$ and CFMM C_2 makes no trade. Overall the trader receives 15 A for 2 B.

But if they sell 2B together, the price is (3/20) A per B; CFMM C_1 trades to final state $(x_A = 1.5, x_B = 0)$ and CFMM C_2 trades to final state $(x_A = 20/3, x_B = 3/2)$. The trader receives 40/3 A for 2 B.

This example shows that path independence is at a conflict with the interests of the CFMM, and it can be satisfied only in special cases.

We now move on to solution concepts that allow the CFMMs for an extra step after the batch. We show that doing so carefully can expand our design space in interesting and useful ways.

6.4 Beyond Market Equilibria Solutions

Here we study the case where the CFMMs are capable of post-processing their state after the batch trade and before being open to standalone operation. For example, CFMMs can restore price coherence through a careful post-processing step. This can, for example, be facilitated by the batch operator. A simple method is to run a "phantom batch" with no limit orders and an equilibrium computation algorithm implementing Trading Rule U for all CFMMs. By doing so, the batch exchange may retain some desirable properties of solutions that do not guarantee PC in the *main batch* (for example Trading Rule S), and then also attain PC before subsequent operations by running the phantom batch. This 2-step solution seems to violate our impossibility result of Theorem 6.1.6, but it does not! This solution violates the axiom of uniform valuations (Axiom 6.2.2) since the asset valuations in the phantom batch may deviate from those in the main batch.

We showed here how participating CFMMs in a state of cyclic arbitrage can *jointly* post-process from an arbitrary initial state to reach a price coherent state. Importantly, this operation can be performed after adopting *any* equilibrium computation algorithm in the main batch. Our second result in this section is even more interesting. We now study a class of locally computable CFMM trading rules with the following property. Batches where the equilibrium computation algorithm implements a trading rule from this class for all CFMMs, assure that the CFMMs can *locally* post-process and attain a state of JPD. Importantly, this local post-processing strategy does not entail making a trade or changing the CFMM trading function, it simply entails capturing a part of the CFMM reserves as 'profit' and removing it from the market-making pool. In the process, the CFMM reaches a state where the value of the trading function is equal to its pre-batch value. The caveat is that this 2-step process violates asset conservation since assets are effectively removed from the system at the end of it. For this, we first define Trading Rules E and F.

For a standalone CFMM trading in two assets A and B, under Assumption 5.2.2, there is an injective map from the spot price to the CFMM state. Motivated by the 'rebalancing' strategy of Miilionis et. al [244], a CFMM trading rule can be designed to buy as much of asset A as it would hold 'on the curve' when the spot price equals the batch price.

Definition 6.4.1. *For the initial CFMM state x and its trading function f, denote the map from the spot price p to the amount of asset A in the CFMM state by $\mathbb{A}(x, f, p)$, i.e.,*

$$\mathbb{A}(x, f, p) = \{z_A | f(z) = f(x); \frac{\partial f}{\partial x_A}(z) / \frac{\partial f}{\partial x_B}(z) = p\}.$$

Definition 6.4.2. *For the initial CFMM state x and its trading function f, denote the map from the spot price p to the amount of asset B in the CFMM state by $\mathbb{B}(x, f, p)$, i.e.,*

$$\mathbb{B}(x, f, p) = \{z_B | f(z) = f(x); \frac{\partial f}{\partial x_A}(z)/\frac{\partial f}{\partial x_B}(z) = p\}.$$

For strictly quasiconcave CFMM trading functions, $\mathbb{A}(x, f, p)$ and $\mathbb{B}(x, f, p)$ are singleton. For non-strict quasiconcave CFMM trading functions, we can use any arbitrary elements of the sets defined above for the rest of the discussion. This enables us to define two CFMM trading rules.

Definition 6.4.3 (Trading Rule E)**.** *The allocation obtained by a CFMM in market equilibrium is such that it gets as much A as it holds on the intital level curve of f when the spot price equals the batch price. That is: $F_E(x, f, p) = z$ where $z_A = \mathbb{A}(x, f, p_A/p_B)$ and $z_B = (1/p_B)(x_A p_A + x_B p_B - \mathbb{A}(x, f, p_A/p_B)p_A)$.*

In a similar vein, we also have Trading Rule F.

Definition 6.4.4 (Trading Rule F)**.** *The allocation obtained by a CFMM in market equilibrium is such that it gets as much B as it holds on the intital level curve of f when the spot price equals the batch price. That is: $F_F(x, f, p) = z$ where $z_A = (1/p_A)(x_A p_A + x_B p_B - \mathbb{B}(x, f, p_A/p_B)p_B)$ and $z_B = \mathbb{B}(x, f, p_A/p_B)$.*

Observe that both Trading Rules E and F satisfy LCR. We illustrate these trading rules in Figure 6.1. We define a class of LCR trading rules for 2-asset CFMMs – *Strict-Surplus* Trading Rules – as a parameterized interpolation between Trading Rules E and F.

Definition 6.4.5 (Strict-Surplus Trading Rules)**.** *The allocation obtained by a CFMM in market equilibrium under Strict-Surplus Trading Rule with parameter $\theta \in [0, 1]$ is $F_\theta(x, f, p) = z$ where*
$z_A = \theta\mathbb{A}(x, f, p_A/p_B) + (1 - \theta)(1/p_A)(x_A p_A + x_B p_B - \mathbb{B}(x, f, p_A/p_B)p_B)$ and,
$z_B = \theta(1/p_B)(x_A p_A + x_B p_B - \mathbb{A}(x, f, p_A/p_B)p_A) + (1 - \theta)\mathbb{B}(x, f, p_A/p_B)$.

Strict-Surplus Trading Rules have a special property captured in the following theorem.

Theorem 6.4.6. *A CFMM with trading function f and initial state x trading in assets A and B with a Strict-Surplus Trading Rule with parameter θ can, post-batch, extract a non-negative surplus e where*
$e_A = (1 - \theta)(x_A - \mathbb{A}(x, f, p_A/p_B) + (x_B - \mathbb{B}(x, f, p_A/p_B))p_B/p_A)$, and
$e_B = \theta((x_A - \mathbb{A}(x, f, p_A/p_B))p_A/p_B + x_B - \mathbb{B}(x, f, p_A/p_B))$,
and reach a state z' where $z'_A = \mathbb{A}(x, f, p_A/p_B)$ and $z'_B = \mathbb{B}(x, f, p_A/p_B)$.
After surplus extraction, the trading function becomes equal to its pre-batch value, i.e., $f(z') = f(x)$.
If all CFMMs in the batch extract their respective surplus, the final state satisfies JPD.

Proof. Follows from Definitions 6.4.1, 6.4.2, and 6.4.5. □

This result augments our design space in a significant manner if the batch exchange believes that JPD is important but wants to try solution concepts other than Trading Rule U. Informally, Trading Rule F, which is a Strict Surplus trading rule, may provide more liquidity to the market than Trading Rule U in some cases (observe from Fig. 6.1 that in this example, Trading Rule F implies a larger trade for the CFMM than Trading Rule U at the same batch price).

6.5 A Convex Program for 2-asset WGS Demands

Here we give a convex program that computes market equilibria in batch exchanges that incorporate CFMMs that trade between two assets and use locally-computable trading rules that satisfy WGS. Alternatively, this program computes equilibria in Arrow-Debreu exchange markets where every agent's demand response satisfies WGS, and every agent has utility in only two assets. The program is based on the convex program of Devanur et al. [146] for linear exchange markets.

The key observation is that 2-asset CFMMs satisfying WGS can be viewed as (uncountable) collections of agents with linear utilities and infinitesimal endowments. This correspondence lets us replace a summation over agents with an integral over this collection of agents. However, proving the correctness of our program requires direct analysis of the objective instead of the argument based on Lagrange multipliers used in [146]. The objective is smooth on the feasible region, and gradients are easily computable for many natural CFMMs.

6.5.1 From a Demand Function to a Continuum of Limit Orders

Suppose that a batch participant is only interested in two assets A and B. We first give a viewpoint that corresponds such an agent to a continuum of exchange market agents with linear utility functions that trade between A and B. Under LCR, such an agent's demand function can only depend on the price between the two assets (that is, $r_{AB} = \frac{p_A}{p_B}$). We can therefore define a function $S_A(\cdot)$ that gives, for any price r_{AB}, the amount of asset A that the agent sells (in net, relative to its initial endowment).

Definition 6.5.1 (Supply)**.** *Consider a batch participant with endowment x that only trades between assets A and B. The Supply of the participant of asset $\mathcal{A}$ at exchange rate $r_{AB} > 0$ is the set $S_{\mathcal{A}}(r_{AB}) = \{\max(0, x_{\mathcal{A}} - z_{\mathcal{A}})\}$, where z is an amount of $\mathcal{A}$ that could be held by the agent after a batch is settled.*

For agents maximizing a utility function (i.e. limit sell orders), z ranges over utility-maximizing allocations, and for a CFMM using some trading rule, z is the allocation specified by the trading rule.

A supply function $S_A(\cdot)$ is *monotonic* if, for any $r_{AB} < \hat{r}_{AB}$ and $z_1 \in S_A(r_{AB})$, $z_2 \in S_A(\hat{r}_{AB})$, we have $z_1 \leq z_2$.

For every exchange rate r_{AB}, it must be the case that either $0 \in S_{\mathcal{A}}(r_{AB})$ or $0 \in S_{\mathcal{B}}(1/r_{AB})$. In the rest of this section, it will be convenient to consider each density function separately. Specifically, for the purpose of the convex program, we will assume that an exchange market consists of a set of supply functions, and that each supply function i sells good $\mathcal{A}_i$ and buys $\mathcal{B}_i$ (and therefore write only $S_i(\cdot)$, when clear from context).

When a CFMM trades based on a locally-computable trading rule, then $S_i(\cdot)$ outputs a single point. Similarly, for an agent maximizing a utility function, $S_i(\cdot)$ always outputs a single point when the utility function is strictly quasi-concave. Many natural locally-computable CFMM trading rules, such as rules S and U correspond to differentiable supply functions when CFMM trading functions are strictly quasi-concave.

Proving existence of equilibria requires that each $S_i(\cdot)$ has a closed graph (for an application of Kakutani's fixed-point theorem). This requirement holds for trading rules S and U when trading functions are quasi-concave.

Definition 6.5.2 (Inverse Supply)**.** *$S_i^{-1}(t)$ is the least upper bound on the set $\{r|S_i(r) \leq t\}$. When the set is empty, $S_i^{-1}(t) = 0$, and is ∞ when the set is unbounded.*

We make the following simplifying assumption in the rest of the discussion.

Assumption 6.5.3. *Every Supply function $S_i(\cdot)$ is either a monotonic, differentiable, point-valued function with $S_i(\infty) > 0$ or a "threshold" function of following form: for some constants $r_i \geq 0$ and $s_i > 0$, $S_i(r) = 0$ for all $r < r_i$, $S_i(r) = s_i$ for $r > r_i$, and $S_i(r_i) = [0, s_i]$.*

The results below only require that an agent's net trading behavior at an equilibrium is expressible as a sum of finitely many supply functions trading from one asset to another. [13]

Continuous, strictly quasi-concave trading functions give point-valued, differentiable supply functions, and linear utility functions give threshold supply functions. Each limit sell order, therefore, corresponds to a threshold supply function.

When supply functions are differentiable, we can define the marginal supply of a CFMM at a given exchange rate.

Definition 6.5.4 (Marginal Supply function)**.** *The marginal supply function $s(r)$ of an agent selling $\mathcal{A}$ in exchange for $\mathcal{B}$ is $\frac{\partial S(r)}{\partial r}$.*

The marginal supply function represents the marginal amount of $\mathcal{A}$ that an agent sells at each price. Informally, $s(r)$ denotes the size of a limit sell offer with minimum price r, and an agent with supply function $S(r)$ behaves as a continuum of these marginal limit sell offers.

[13] The expression may not be unique. Additionally, if one agent's behavior is expressible as two threshold functions, one from A to B with limit price r and another from B to A with limit price $1/r$, then when $\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}} = r$, these supply functions may trade with each other, in net. This does not affect any results.

Supply functions are monotonic if and only if an agent's behaviour satisfies WGS.

Lemma 6.5.5. *The behavior of a CFMM with a differentiable supply function satisfies WGS if and only if $S_i(\cdot)$ is monotonic.*

Proof. If there is some r such that $S(r)$ is strictly decreasing at $r = p_\mathcal{A}/p_\mathcal{B}$, then there is some $p_\mathcal{B}'$ such that $p_\mathcal{B}' < p_\mathcal{B}$ (so $r' = p_\mathcal{A}/p_\mathcal{B} > r$), $S_i(r) > S_i(r')$, and $S_i(\cdot) > 0$ on the interval $[r, r']$ (so the other asset's supply function is 0 on this interval). In other words, a decrease in the valuation of $\mathcal{B}$ causes the agent to sell less $\mathcal{A}$, which means that its demand for $\mathcal{A}$ increases and violates WGS.

Conversely, if $S_i(r)$ is always nondecreasing, then for every $r = p_\mathcal{A}/p_\mathcal{B}$ and $r' = p_\mathcal{A}/p_\mathcal{B}'$ with $p_\mathcal{B}' < p_\mathcal{B}$ (so $r' > r$), $S_i(r') \geq S_i(r)$. In other words, the agent's demand for $\mathcal{A}$ never increases as the valuation of $\mathcal{B}$ falls, satisfying WGS. $\square$

Corollary 6.5.6. *All two-asset CFMMs with trading functions satisfying Assumption 5.2.2 have a WGS demand function under Trading Rule S.*

Proof. Trading functions satisfying Assumption 5.2.2 are nondecreasing in every asset. At exchange rate r, the CFMM with endowment x sells $S(r)$ units of $\mathcal{A}$ to receive $rS(r)$ units of $\mathcal{B}$. At $r' > r$, the CFMM has sufficient budget to purchase $r'S(r)$ units of B, and if it does not purchase at least this much, then the trading function cannot be nondecreasing from $(x_\mathcal{A} - S(r), x_\mathcal{B} + rS(r))$ to $(x_\mathcal{A} - S(r), x_\mathcal{B} + r'S(r))$ $\square$

6.5.2 Convex Program

We assume that a set of agents has utility functions that imply a set of supply functions that satisfy Assumption 6.5.3, with $S_i(\cdot)$ for $i \in \mathcal{J}$ continuous and $S_i(\cdot)$ for $i \in \mathcal{K}$ threshold functions. These agents trade a set of assets $\mathfrak{A}$.

Variables $\{p_\mathcal{A}\}$ denote the valuation of assets $\mathcal{A} \in \mathfrak{A}$. The quantity $y_i/p_{\mathcal{A}_i}$ denotes the amount of good $\mathcal{A}_i$ that supply function i sells to the market, receiving $y_i/p_{\mathcal{B}_i}$ units of $\mathcal{B}_i$. By construction, at equilibrium, it must be the case that $y_i/p_{\mathcal{A}_i} \in S_i(\frac{p_\mathcal{A}}{p_\mathcal{B}})$. Define $\beta_{i,r}(p) = \min(p_{\mathcal{A}_i}, p_{\mathcal{B}_i} r)$. Informally, $\beta_{i,r}(p)$ is the inverse best bang-per-buck for the marginal limit sell order at limit price r of supply function i.

Finally, for continuous marginal supply functions, define $g_i(t)$ to be $\int_0^{S_i^{-1}(t)} s_i(r) \ \ln(1/r) \ dr$. For threshold supply functions, define $g_i(t) = \min(s_i, t) \ln(1/r_i)$.

Lemma 6.5.7. *$g_i(\cdot)$ is a concave function, and $-p_{\mathcal{A}_i} g_i(y_i/p_{\mathcal{A}_i})$ is convex.*

Proof. For continuous supply density functions,

$$\begin{aligned}\frac{\partial}{\partial t} g_i(t) &= \frac{\partial}{\partial t}\int_0^{S_i^{-1}(t)} s_i(r)\ln(1/r)dr \\ &= d(S_i^{-1}(t))\ln\left(\frac{1}{S_i^{-1}(t)}\right)\frac{1}{d(S_i^{-1}(t))} \\ &= \ln\left(\frac{1}{S_i^{-1}(t)}\right)\end{aligned}$$

Because the derivative of $g_i(\cdot)$ is a decreasing function, $g_i(\cdot)$ must be concave (for $t \geq 0$). Concavity clearly holds for threshold supply functions.

$-p_{\mathcal{A}_i} g_i(y_i/p_{\mathcal{A}_i})$ is the perspective transformation of a convex function, so it is convex. □

Altogether, we get the following convex program:

Theorem 6.5.8. *The following program is convex and always feasible. Its objective value is always non-negative. When the objective value is* 0*, the solution forms an exchange market equilibrium with nonzero prices, and when such an equilibrium exists, the minimum objective value is* 0*.*

$$\begin{aligned}&\textit{Minimize} \sum_{i\in\mathcal{J}} p_{\mathcal{A}_i}\int_0^\infty \left(s_i(r)\ \ln\left(\frac{p_{\mathcal{A}_i}}{\beta_{i,r}(p)}\right)\right)\ dr && \text{(COP)}\\ &\qquad + \sum_{i\in\mathcal{K}} p_{\mathcal{A}_i} s_i \ln\left(\frac{p_{\mathcal{A}_i}}{\beta_{i,r_i}(p)}\right) - \sum_{i\in\mathcal{J}\cup\mathcal{K}} p_{\mathcal{A}_i}\ g_i(y_i/p_{\mathcal{A}_i}) \\ &\textit{Subject to} \sum_{i\in\mathcal{J}\cup\mathcal{K}:\mathcal{A}_i=\mathcal{C}} y_i = \sum_{i\in\mathcal{J}\cup\mathcal{K}:\mathcal{B}_i=\mathcal{C}} y_i \qquad \forall\mathcal{C}\in\mathfrak{A} && \text{(C1)}\\ &\qquad p_{\mathcal{C}} \geq 1 \qquad \forall\mathcal{C}\in\mathfrak{A} && \text{(C2)}\\ &\qquad 0 \leq y_i \leq p_{\mathcal{A}_i} S_i(\infty) \qquad \forall i\in\mathcal{J}\cup\mathcal{K}. && \text{(C3)}\end{aligned}$$

Proof. Lemma 6.5.9 shows the convexity and feasibility of the program. Lemma 6.5.10 shows that the objective value is nonnegative, and is 0 if and only if the optimal solutions satisfy $y_i/p_{\mathcal{A}_i} \in S_i(p)$.

Given the mapping between agents in the Arrow-Debreu exchange market and supply functions in the convex program, as in Assumption 6.5.3, each y_i implies a transfer of $y_i/p_{\mathcal{A}_i}$ units of $\mathcal{A}_i$ out of the corresponding agent's endowment and $y_i/p_{\mathcal{B}_i}$ units of $\mathcal{B}_i$ into their endowment. By construction of the density functions, these transfers give, for each agent, an optimal bundle of assets in the original exchange market. When an equilibrium exists, the optimal objective value is 0. □

Lemma D.1.3 in the appendix shows that an equilibrium with positive prices exists, given mild assumptions (i.e. Assumption D.1.1).

Lemma 6.5.9. *The program with objective COP and constraints C1, C2, and C3 is convex and feasible.*

Proof. $\beta_{i,r}(\cdot)$ is concave, and $\ln(\cdot)$ is concave and nondecreasing, so each term $s_i(r)\ln(p_{\mathcal{A}i}/\beta_{i,r}(p))$ is convex, and the integral or sum of convex functions is convex. Feasibility follows from setting $y_i = 0$ for all i and and $p_{\mathcal{C}} = 1$ for all $\mathcal{C} \in \mathfrak{A}$. □

As compared to the program of [146], we combining the first and second constraints of that program (avoiding infeasibility when an equilibrium does not exist) and implicitly upper bound the $\{y_i\}$ variables through the utility calculations in the $\{g_i(\cdot)\}$ functions.

Lemma 6.5.10. *The objective value of the convex program is nonnegative, and is zero if and only if $y_i \in p_{\mathcal{A}i}S_i(p_{\mathcal{A}i}/p_{\mathcal{B}i})$ for all $i \in \mathcal{J} \cup \mathcal{K}$.*

Proof. Consider the following three quantities.

1. $E_1 = \sum_{i\in\mathcal{J}} p_{\mathcal{A}i} \int_0^\infty \left(s_i(r)\ \ln\left(\frac{p_{\mathcal{A}i}}{\beta_{i,r}(p)}\right)\right) dr + \sum_{i\in\mathcal{K}} p_{\mathcal{A}i} s_i \ln\left(\frac{p_{\mathcal{A}i}}{\beta_{i,r_i}(p)}\right)$
2. $E_2 = \sum_{i\in\mathcal{J}} p_{\mathcal{A}i} \int_0^{S_i^{-1}(y_i/p_{\mathcal{A}i})} \left(s_i(r)\ \ln\left(\frac{p_{\mathcal{A}i}}{\beta_{i,r}(p)}\right)\right)\ dr + \sum_{i\in\mathcal{K}} y_i \ln\left(\frac{p_{\mathcal{A}i}}{\beta_{i,r_i}(p)}\right)$
3. $E_3 = \sum_{i\in\mathcal{J}\cup\mathcal{K}} p_{\mathcal{A}i}\, g_i(y_i/p_{\mathcal{A}i}) = \sum_{i\in\mathcal{J}} p_{\mathcal{A}i} \int_0^{S_i^{-1}(y_i/p_{\mathcal{A}i})} s_i(r)\ \ln(1/r)\ dr + \sum_{i\in\mathcal{K}} p_{\mathcal{A}i} \min(s_i, y_i/p_{\mathcal{A}i}) \ln(1/r_i)$

By construction, for all r, $\ln\beta_{i,r}(p) \le \ln(r) + \ln(p_{\mathcal{B}i})$, or equivalently, $\ln(1/r) \le -\ln(\beta_{i,r}(p)) + \ln(p_{\mathcal{B}i})$. Furthermore, note that by the constraint (C1), for all assets $\mathcal{C}$, $\sum_{i:\mathcal{A}_i=\mathcal{C}} y_i \ln p_{\mathcal{C}} = \sum_{i:\mathcal{B}_i=\mathcal{C}} y_i \ln p_{\mathcal{C}}$. Additionally, for all $i \in \mathcal{K}$, because $y_i \le s_i p_{\mathcal{A}i}$, it must be the case that $\min(s_i, y_i/p_{\mathcal{A}i}) = y_i/p_{\mathcal{A}i}$.

Using these facts, we get

$$\begin{aligned}
E_3 &= \sum_{i\in\mathcal{J}} p_{\mathcal{A}i} \int_0^{S_i^{-1}(y_i/p_{\mathcal{A}i})} s_i(r)\ \ln(1/r)\ dr + \sum_{i\in\mathcal{K}} p_{\mathcal{A}i} \min(s_i, y_i/p_{\mathcal{A}i}) \ln(1/r_i) \\
&\le \sum_{i\in\mathcal{J}} p_{A_i} \int_0^{S_i^{-1}(y_i/p_{A_i})} s_i(r)\left(-\ln\beta_{i,r}(p) + \ln(p_{\mathcal{B}i})\right)\ dr + \sum_{i\in\mathcal{K}} y_i\left(-\ln\beta_{i,r_i}(p) + \ln(p_{\mathcal{B}i})\right) \\
&= \sum_{i\in\mathcal{J}\cup\mathcal{K}} y_i \ln p_{\mathcal{B}i} + \sum_{i\in\mathcal{J}} p_{A_i} \int_0^{S_i^{-1}(y_i/p_{A_i})} s_i(r)\left(-\ln\beta_{i,r}(p)\right)\ dr + \sum_{i\in\mathcal{K}} y_i\left(-\ln\beta_{i,r_i}(p)\right) \\
&= \sum_{i\in\mathcal{J}\cup\mathcal{K}} y_i \ln p_{\mathcal{A}i} + \sum_{i\in\mathcal{J}} p_{A_i} \int_0^{S_i^{-1}(y_i/p_{A_i})} s_i(r)\left(-\ln\beta_{i,r}(p)\right)\ dr + \sum_{i\in\mathcal{K}} y_i\left(-\ln\beta_{i,r_i}(p)\right) \\
&= \sum_{i\in\mathcal{J}} p_{A_i} \int_0^{S_i^{-1}(y_i/p_{A_i})} s_i(r)\left(\ln p_{\mathcal{A}i} - \ln\beta_{i,r}(p)\right)\ dr + \sum_{i\in\mathcal{K}} y_i\left(\ln p_{\mathcal{A}i} - \ln\beta_{i,r_i}(p)\right) \\
&= E_2
\end{aligned}$$

For any i, define $\hat{r}_i = \frac{p_{\mathcal{A}i}}{p_{\mathcal{B}i}}$.

For any $r < \hat{r}_i$, $p_{\mathcal{B}i}/\beta_{i,r}(p) = 1/r$, and otherwise $p_{\mathcal{B}i}/\beta_{i,r}(p) = 1/\hat{r}_i \ge 1/r$. As such, for any $i \in \mathcal{J}$, $\int_0^{S_i^{-1}(y_i/p_{\mathcal{A}i})} s_i(r)\ \ln(1/r)\ dr = \int_0^{S_i^{-1}(y_i/p_{\mathcal{A}i})} s_i(r)\ \ln(p_{\mathcal{B}i}/\beta_{i,r}(p))\ dr$ if and only if $y_i/p_{\mathcal{A}i} \le S_i(p_{\mathcal{A}i}/p_{\mathcal{B}i})$. Similarly, for any $i \in \mathcal{K}$, $p_{\mathcal{A}i}\min(s_i, y_i/p_{\mathcal{A}i})\ln(1/r_i) = y_i\left(-\ln\beta_{i,r_i}(p) + \ln(p_{\mathcal{B}i})\right)$ if and only if $y_i = 0$ if $r_i > \hat{r}_i$. These conditions hold for each $i \in \mathcal{J} \cup \mathcal{K}$ if and only if $E_2 = E_3$.

Furthermore, observe that $p_{\mathcal{A}i}/\beta_{i,r}(p) \geq 1$ for all r, and is equal to 1 for any $r > \hat{r}_i$. Continuing to rearrange E_2 gives

$$\begin{aligned} E_2 &= \sum_{i \in \mathcal{J}} p_{\mathcal{A}i} \int_0^{S_i^{-1}(y_i/p_{\mathcal{A}_i})} s_i(r) \ln \left(\frac{p_{\mathcal{A}i}}{\beta_{i,r}(p)} \right) \, dr + \sum_{i \in \mathcal{K}} y_i \ln \left(\frac{p_{\mathcal{A}i}}{\beta_{i,r_i}(p)} \right) \\ &\leq \sum_{i \in \mathcal{J}} p_{\mathcal{A}i} \int_0^{\infty} \left(s_i(r) \ \ln \left(\frac{p_{\mathcal{A}i}}{\beta_{i,r}(p)} \right) \right) \, dr + \sum_{i \in \mathcal{K}} p_{\mathcal{A}i} s_i \ln \left(\frac{p_{\mathcal{A}i}}{\beta_{i,r_i}(p)} \right) \\ &= E_1 \end{aligned}$$

Observe that for any $i \in \mathcal{J}$, $\int_0^{S_i^{-1}(y_i/p_{\mathcal{A}_i})} s_i(r) \left(\ln p_{\mathcal{A}i} - \ln \beta_{i,r}(p)\right) \, dr = \int_0^{\infty} s_i(r) \left(\ln p_{\mathcal{A}i} - \ln \beta_{i,r}(p)\right) \, dr$ if and only if $y_i/p_{\mathcal{A}i} = S_i(p_{\mathcal{A}i}/p_{\mathcal{B}i})$. Additionally, for any $i \in \mathcal{K}$, if $r_i < \hat{r}_i$, then $y_i \left(\ln p_{\mathcal{A}i} - \ln \beta_{i,r_i}(p)\right) = p_{\mathcal{A}i} s_i \ln \left(\frac{p_{\mathcal{A}i}}{\beta_{i,r_i}(p)} \right)$ if and only if $y_i = p_{\mathcal{A}} s_i$. These conditions hold for each $i \in \mathcal{J} \cup \mathcal{K}$ if and only if $E_1 = E_2$.

As such, $E_1 \geq E_3$, and the inequality is tight if and only if $y_i/p_{\mathcal{A}i} \in S_i(p_{\mathcal{A}i}/p_{\mathcal{B}i})$ for all $i \in \mathcal{J} \cup \mathcal{K}$. But the objective of the convex program is $E_1 - E_3$, proving the theorem. □

6.5.3 Rationality of Convex Program

The program of [146] always has a rational solution. Our program may not, given certain CFMMs. However, rational solutions exist when CFMMs belong to the following class.

Theorem 6.5.11. *If the expression $p_{\mathcal{A}i} S_i(p_{\mathcal{A}_i}/p_{\mathcal{B}_i})$ is a piecewise-linear, rational function of $p_{\mathcal{A}i}$ and $p_{\mathcal{B}i}$ for all i, on the range where $S_i(\cdot) > 0$ and $S_i(\cdot)$ is monotonic, then the convex program has an optimal rational solution.*

Proof. Note that it suffices without loss of generality to consider functions only of the forms outlined in Assumption 6.5.3. For each continuous $S_i(\cdot)$, at an optimal point $(\hat{p}, \hat{y})$, for every CFMM i, it must be the case that $\hat{p_{\mathcal{A}i}} S_i(\hat{p_{\mathcal{A}i}}/\hat{p_{\mathcal{B}i}}) = \hat{y}_i$.

Then there are two linear functions $q_i^+(\cdot)$, $q_i^-(\cdot)$ such that, on an open neighborhood about $\hat{y}$, $y_i = q_i^+(p_{A_i}, p_{B_i})$ when $\frac{p_{\mathcal{A}i}}{p_{\mathcal{B}i}} \geq \frac{\hat{p_{\mathcal{A}i}}}{\hat{p_{\mathcal{B}i}}}$ and $y_i = q_i^-(p_{A_i}, p_{B_i})$ otherwise.

To the set of existing constraints in the convex program, add the constraints that $y_i = q_i^+(p_{\mathcal{A}i}, p_{\mathcal{B}i})$ and $y_i = q_i^-(p_{\mathcal{A}i}, p_{\mathcal{B}i})$.

For each $S_i(\cdot)$ representing a threshold function of size s_i at exchange rate r_i, if $\hat{y}_i = s_i$, add the constraint that $y_i = s_i$, and if $\hat{y}_i = 0$, add the constraint that $p_{\mathcal{A}i} = r_i p_{\mathcal{B}i}$.

This system of constraints is clearly satisfiable, and every point satisfying the constraints is a market equilibrium (every point satisfies $y_i \in p_{\mathcal{A}i} S_i(p_{\mathcal{A}i}/p_{\mathcal{B}i})$). Each of the constraints is linear and rational, so these constraints define a rational polytope. The extremal points of this polytope must therefore be rational. □

Example 6.5.12. *The supply function of the constant product CFMM with reserves $(x_{\mathcal{A}}, x_{\mathcal{B}})$ under Trading Rule U in a batch exchange is* $\max(0, (r x_{\mathcal{A}} - x_{\mathcal{B}})/(2r))$.

Proof. Without loss of generality, assume batch price r (units of $\mathcal{B}$ per $\mathcal{A}$) is greater than the CFMM's pre-batch spot price of $x_{\mathcal{B}}/x_{\mathcal{A}}$, so the CFMM in net sells $\mathcal{A}$ to the market and purchases $\mathcal{B}$. The CFMM, under Trading Rule U, makes a trade so that its reserves $(z_{\mathcal{A}}, z_{\mathcal{B}})$ after trading satisfy the following two conditions.

First, the post-batch spot price of the CFMM, $z_{\mathcal{B}}/z_{\mathcal{A}}$, must be r. And second, the CFMM must trade at the batch exchange rate, so $(x_{\mathcal{A}} - z_{\mathcal{A}})r = (z_{\mathcal{B}} - x_{\mathcal{B}})$. Note that $S(r) = x_{\mathcal{A}} - z_{\mathcal{A}}$.

Combining these equations gives

$$\frac{x_{\mathcal{B}} + rS(r)}{x_{\mathcal{A}} - S(r)} = r.$$

Solving for $S(r)$ gives

$$S(r) = \frac{rx_{\mathcal{A}} - x_{\mathcal{B}}}{2r}.$$

□

However, this convex program cannot always have rational solutions; in fact, there exist simple examples using natural utility functions for which the program has only irrational solutions.

Example 6.5.13. *There exists a batch instance containing one CFMM based on the logarithmic market scoring rule and one limit sell offer that only admits irrational equilibria when the CFMM implements Trading Rule U.*

Proof. Consider a batch instance trading assets A and B that contains one CFMM and one limit sell order. The CFMM uses trading function $f(x) = -(e^{-x_A}+e^{-x_B})+2$, with initial state $x_{A0} = x_{B0} = 1$. The limit sell order is to sell 100 units of A for B, with a minimum price of $\frac{1}{2}$ B per A.

If the batchprice $p = p_A/p_B$ is strictly greater than $\frac{1}{2}$, then the limit sell order must sell the entirety of its A to receive at least $100p > 50$ units of B, which the CFMM cannot provide.

On the other hand, if the batch price is less than $\frac{1}{2}$, then the limit sell order will not sell any A but the CFMM demands a nonzero amount of A. Thus, at equilibrium, the batch price equals $\frac{1}{2}$.

Let the limit sell order sell τ units of A. Clearly, in equilibrium, $0 \leq \tau \leq 100$.

Furthermore, the spot price of the CFMM at equilibrium must be equal to $\frac{1}{2}$. The spot exchange rate of this CFMM is

$$\frac{\frac{\partial f}{\partial x_A}}{\frac{\partial f}{\partial x_B}} = \frac{e^{-x_A}}{e^{-x_B}} = e^{-x_A+x_B}$$

Thus, at equilibrium, we must have that

$$p = e^{-(x_{A0}+\tau)+(x_{B0}-p\tau)} = e^{-x_{A0}+x_{B0}}e^{-\tau-p\tau} = e^{-\tau(1+p)}.$$

This gives $e^{-\frac{3\tau}{2}} = \frac{1}{2}$. That is, $\tau = \frac{2}{3}\ln(2)$, which is irrational. □

6.6 Conclusion and Open Problems

Constant Function Market Makers have become some of the blockchain ecosystem's most widely used exchange systems. Batch trading has been proposed and deployed to combat some shortcomings of decentralised and traditional exchanges. Different implementations in practice have taken substantially different approaches to how these two innovations should interact.

We develop an axiomatic framework and describe several desirable properties of the combined system. While several pairs of these properties cannot be guaranteed simultaneously, we are able to provide algorithms that achieve subsets of these properties.

Finally, we construct a convex program that computes equilibria on batches containing CFMMs that each trade only two assets. For many natural classes of CFMMs, the objective of this convex program is smooth, and the program has rational solutions.

6.6.1 Open Problems

An important question for batch exchange deployments is the fee structure for CFMMs. Finding fair compensation for liquidity provision in batch exchanges may require, for example, quantitative models and a careful analysis of existing trade data.

Giving a convex program for exchange market equilibria for general WGS utilities (not just on two assets) continues to be an important open problem.

In this work we find Pareto optimal solutions only in a special case of two-asset CFMMs, when the CFMMs are price-coherent pre-batch. Designing algorithms for finding Pareto optimal solutions in multi-asset batches in polynomial time is an important open problems. We show that such an algorithm will necessarily have to give up LCR for CFMMs (Theorem 6.1.7).

From an applied perspective, iterative algorithms like Tâtonnement [127], if implemented naïvely, would require for each demand query an iteration over every CFMM. Chapter 3 gives a preprocessing step to enable computation of the aggregate response of a group of limit sell offers in logarithmic time (§3.5.1); identifying a subclass of CFMM trading functions that admit an analogous preprocessing step or, more generally, some way of computing the aggregate demand response of a large group of CFMMs in sublinear time would be of great practical use for batch exchanges using these types of algorithms.

Chapter 7

Conclusions

If digital currencies and a new generation of digital financial systems are to achieve their full potential in any context, they must be built on infrastructure that is scalable, flexible, and efficient.

From a computational perspective, chapter 2 develops a new architectural paradigm that ensures near-linear computational scalability in a replicated state machine underlying digital currencies. A combination of commutativity in transaction semantics and the choice to order blocks of transactions only partially, instead of totally ordering all transactions, ensures even (and high) performance on both low-contention (best-case) and high-contention (worst-case) transaction workloads. These systems achieve this scalability without substantially limiting transaction flexibility or forcing negative externalities onto end-users. And chapter 4 completes the picture for the rare class of applications which cannot be rewritten into this commutative model, recovering a total ordering between transactions that reflects the way that information propagates nonuniformly through a distributed system.

Additionally, chapter 3 demonstrates the practical feasibility of implementing an asset exchange within this architectural paradigm that not only achieves the same computational scalability properties but also provides novel economic efficiency and fairness properties. As the semantics of a traditional continuous double-auction mechanism are far from commutative, this requires using instead a uniform-price batch exchange mechanism between many assets, and instantiating this at scale requires computing market equilibria efficiently at scale.

Subsequent chapters then extend the study of economic mechanisms for exchanging digital currencies—chapter 5 studies the fine-grained tradeoffs within a class of automated mechanisms known as "CFMMs," and chapter 6 explores the different economic consequences of the many plausible ways of combining CFMMs into batch exchange mechanisms.

Of course, the optimal computational and economic design for infrastructure for anything related to digital currencies must vary based on context, requirements, and policy objectives. However, I hope that the research of this thesis can push the boundary of what is thought to be possible, help policymakers make well-informed choices, and ultimately help build infrastructure for digital money that can last for decades.

Appendix A

Chapter 3 Supporting Information

A.1 Mathematical Model Underlying SPEEDEX

Mathematically, SPEEDEX relies on a correspondence between a batch of trade offers and an instance of a linear Arrow-Debreu Exchange Market [78]. Specifically, SPEEDEX's batch computation is equivalent to the problem of computing equilibria in these markets.

A.1.1 Arrow-Debreu Exchange Markets

The Arrow-Debreu Exchange Market is a classic model from the economics and theoretical computer science literature. Conceptually, there exists in this market a set of independent *agents*, each with its own *endowment* of goods. Each agent has some set of preferences over possible collections of goods. These goods are tradeable on an open market, and agents, all at the same time, make any set of trades that they wish with *the market* (or *auctioneer*), not directly with each other.

Definition A.1.1 (Arrow-Debreu Exchange Market)**.** *An Arrow-Debreu Exchange Market consists of a set of goods* $\mathfrak{A}$ *and a set of agents* $j \in \{1, ..., M\}$*. Every agent* j *has a utility function* $u_j(\cdot)$ *and an endowment* $e_j \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$*.*

When the market trades at prices $p \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$*, every agent sells their endowment to the market in exchange for revenue* $s_j = p \cdot e_j$*, which the agent immediately spends at the market to buy back an optimal bundle of goods* $x_j \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$ *- that is,* $x_j = \arg\max_{x : \sum_{\mathcal{A} \in \mathfrak{A}} x_{\mathcal{A}} p_{\mathcal{A}} \leq s_j} u_j(x)$*.*

There are countless variants on this definition. Typically the utility functions are assumed to be quasi-convex. Some variants include stock dividents, corporations, production of new goods from existing goods, and multiple trading rounds. SPEEDEX uses only the model outlined above—SPEEDEX looks only at snapshots of the market, i.e., once per block, and computes batch results for each block independently.

One potential objection to the above definition is that it assumes that the abstract market has sufficient quantities available so that every agent can make its preferred trades. We say that a market is at *equilibrium* when agents can make their preferred trades and the market does not have a deficit in any good.

Definition A.1.2 (Market Equilibrium)**.** *An equilibrium of an Arrow-Debreu market is a set of prices p and an allocation x_j for every agent j, such that for all goods $\mathcal{A}$, $\sum_j e_{\mathcal{A},j} \geq \sum_j x_{\mathcal{A},j}$, and x_j is an optimal bundle for agent j. The inequality for asset $\mathcal{A}$ is tight whenever $p_{\mathcal{A}}$ is nonzero.*

Note that an equilibrium includes both a set of market prices and a choice of a utility-maximizing set of goods for each agent. Say, for example, there are two goods $\mathcal{A}$ and $\mathcal{B}$, and one unit of each is sold by other agents to the market. If two agents are indifferent to receiving either good, then the equilibrium must specify whether the first receives $\mathcal{A}$ or $\mathcal{B}$, and vice versa for the second. It would not be a market equilibrium for both of these agents to purchase a unit of $\mathcal{A}$ and no units of $\mathcal{B}$.

A.1.2 From SPEEDEX to Exchange Markets

SPEEDEX users do not submit abstract utility functions to an abstract market. However, most natural types of trade offers can be encoded as a simple utility function.

Specifically, our implementation of SPEEDEX accepts limit sell offers (Definition 6.1.1). In Arrow-Debreu exchange markets, the behavior of a limit sell offer is representable as an agent with a linear utility function.

Observation A.1.3. *Suppose a user submits a sell offer ($\mathcal{S}$, $\mathcal{B}$, e, α). The optimal behavior of this offer (and the user's implicit preferences) is equivalent to maximizing the function $u(x_{\mathcal{S}}, x_{\mathcal{B}}) = \alpha x_{\mathcal{B}} + x_{\mathcal{B}}$ (for $x_{\mathcal{S}}, x_{\mathcal{B}}$ amounts of goods $\mathcal{S}$ and $\mathcal{B}$).*

Proof. Such an offer makes no trades if $p_{\mathcal{S}}/p_{\mathcal{B}} < \alpha$ and trades in full if $p_{\mathcal{S}}/p_{\mathcal{B}} > \alpha$.

The user starts with k units of $\mathcal{S}$. In the exchange market model, the user can trade these k units of $\mathcal{S}$ in exchange for any quantities $x_{\mathcal{S}}$ of $\mathcal{S}$ and $x_{\mathcal{B}}$ of $\mathcal{B}$, subject to the constraint that $p_{\mathcal{S}} x_{\mathcal{S}} + p_{\mathcal{B}} x_{\mathcal{B}} \leq k p_{\mathcal{S}}$.

The function $u(x_{\mathcal{S}}, x_{\mathcal{B}}) = \alpha x_{\mathcal{B}} + x_{\mathcal{S}}$ is maximized, subject to the above constraint, by $(x_B, x_{\mathcal{S}}) = (0, k)$ precisely when $p_{\mathcal{S}}/p_B < \alpha$ and by $(x_{\mathcal{B}}, x_{\mathcal{S}}) = (kp_{\mathcal{S}}/p_{\mathcal{B}}, 0)$ otherwise (and by any convex combination of the two when $p_{\mathcal{S}}/p_{\mathcal{B}} = \alpha$). These allocations correspond exactly to the optimal behavior of a limit sell offer. □

Note that these utility functions have nonzero marginal utility for only two types of assets, and are not arbitrary linear utilities. §6.5 finds anecdotal evidence that this subclass of utility functions may be analytically more tractable than the case of general linear utilities.

A.1.3 Existence of Unique* Equilibrium Prices

Theorem A.1.4. *All of the market instances which SPEEDEX considers contain an equilibrium with nonzero prices.*

Proof. All of the utilities of agents derived from limit sell offers are linear (Observation A.1.3), and have a nonzero marginal utility on the good being sold.

This means our market instances trivially satisfy condition (*) of Devanur et al. [146]. Existence of an equilibrium with nonzero prices follows therefore from Theorem 1 of [146]. □

In fact, all of the equilibria in a market instance contain the same equilibrium prices, unless there are two sets of assets across which no trading activity occurs. In such a case, one might be able to uniformly increase or decrease all the prices together on one set of assets, relative to the other set of assets.

Theorem A.1.5. *Suppose there are two equilibria (p, x) and (p', x') and there exist two assets $\mathcal{A}$ and $\mathcal{B}$ for which $p_\mathcal{A}/p_\mathcal{B} < p'_\mathcal{A}/p'_\mathcal{B}$.*

Then it must be the case that there is a partitioning of the assets $\mathfrak{A}_1, \mathfrak{A}_2$ with $A \in \mathfrak{A}_1, B \in \mathfrak{A}_2$ such that both equilibria include no trading activity across the partition.

Proof. Consider the set of offers trading from $\mathcal{A}$ to $\mathcal{B}$. Let $Z_{\mathcal{A},\mathcal{B}}(r)$ be the set of amounts of asset $\mathcal{A}$ that may be sold (when every agent receives an optimal bundle) by these offers to the market at an exchange rate $r = p_\mathcal{A}/p_\mathcal{B}$. Observe that if $r_1 < r_2$, then every $z_1 \in Z_{\mathcal{A},\mathcal{B}}(r_1)$ is no more than than any $z_2 \in Z_{\mathcal{A},\mathcal{B}}(r_2)$ (as sell offers always prefer higher exchange rates).

At the equilibrium (p, x), let $z_{\mathcal{A},\mathcal{B}}$ be the total amount of $\mathcal{A}$ sold for $\mathcal{B}$ for every asset pair (and $z'_{\mathcal{A},\mathcal{B}}$ similarly for (p', x')). Note that $z_{\mathcal{A},\mathcal{B}} \in Z_{\mathcal{A},\mathcal{B}}(p_\mathcal{A}/p_\mathcal{B})$.

Suppose that there exists a pair of assets $\mathcal{A}, \mathcal{B}$ as in the theorem statement. Then there exists a set of assets $\mathfrak{A}_1$ such that for every asset pair $\mathcal{C} \in \mathfrak{A}_1$ and $\mathcal{D} \notin \mathfrak{A}_1$, $p_\mathcal{C}/p_\mathcal{D} < p'_\mathcal{C}/p'_\mathcal{D}$.

For each of these asset pairs, we must have that $z_{\mathcal{C},\mathcal{D}} \leq z'_{\mathcal{C},\mathcal{D}}$, $z_{\mathcal{D},\mathcal{C}} \geq z'_{\mathcal{D},\mathcal{C}}$, and $\frac{p_\mathcal{C}}{p_\mathcal{D}} z_{\mathcal{C},\mathcal{D}} \leq \frac{p'_\mathcal{C}}{p'_\mathcal{D}} z'_{\mathcal{C},\mathcal{D}}$. Combining these equations gives

$$p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq (p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}}) p_\mathcal{D}/p'_\mathcal{D}$$

Each of these inequalities is tight if and only if $z_{\mathcal{C},\mathcal{D}} = 0$.

It is without loss of generality to rescale p' so that $p_\mathcal{D}/p'_\mathcal{D} < 1$ for all $\mathcal{D} \notin \mathfrak{A}_1$. Thus,

$$p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq (p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}})$$

Because (p, x) and (p', x') are equilibria, we must have that

$$0 = \sum_{\mathcal{C} \in \mathfrak{A}_1} \sum_{\mathcal{D} \notin \mathfrak{A}_1} p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq \sum_{\mathcal{C} \in \mathfrak{A}_1} \sum_{\mathcal{D} \notin \mathfrak{A}_1} p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}}$$

But the second inequality is tight only if each $z_{\mathcal{C},\mathcal{D}} = 0$.

Hence, (p', x') can only be an equilibrium if there exists a partitioning of the assets that separates $\mathcal{A}$ and $\mathcal{B}$, and for which there is no trading activity between the sets in either equilibrium. □

Corollary A.1.6. *Let (p, x) be an equilibrium.*

Construct an undirected graph $G = (V, E)$ with one vertex for each asset, and an edge $e = (\mathcal{A}, \mathcal{B}) \in E$ if, at equilibrium, any $\mathcal{A}$ is sold for $\mathcal{B}$ or any $\mathcal{B}$ is sold for $\mathcal{A}$.

If G is connected, then the market equilibrium prices p are unique (up to uniform rescaling).

Proof. If the theorem hypothesis holds, then for any other equilibrium (p', x'), it must be the case that for every asset pair $(\mathcal{A}, \mathcal{B})$, $p_{\mathcal{A}}/p_{\mathcal{B}} = p'_{\mathcal{A}}/p'_{\mathcal{B}}$. By Theorem A.1.5, if this did not hold, then there would exist a partitioning of V into two sets of assets, across which there is no trading at equilibrium (p, x) (contradicting the assumption that G is connected). □

A.2 Approximation Error

SPEEDEX measures two forms of approximation error: first, every trade is charged a ε transaction commision, and second, some offers with in-the-money limit prices might not be able to be executed (while preserving asset conservation). Formally, the output of the batch price computation is a price $p_{\mathcal{A}}$ on each asset $\mathcal{A}$, and a trade amount $x_{\mathcal{A},\mathcal{B}}$ denoting the amount of $\mathcal{A}$ sold in exchange for $\mathcal{B}$.

Formally, we say that the result of a batch price computation is (ε, μ)-approximate if:

1 Asset conservation is preserved with an ε commission. The amount of $\mathcal{A}$ sold to the auctioneer, $\Sigma_{\mathcal{B}} x_{\mathcal{A},\mathcal{B}}$, must exceed the amount of $\mathcal{A}$ bought from the auctioneer, $\Sigma_{\mathcal{B}}(1-\varepsilon)\frac{p_{\mathcal{B}}}{p_{\mathcal{A}}} x_{\mathcal{B},\mathcal{A}}$.

2 No offer trades outside of its limit price. That is to say, an offer selling $\mathcal{A}$ for $\mathcal{B}$ with a limit price of r cannot execute if $\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}} < r$.

3 No offer with a limit price "far" from the batch exchange rate does not trade. That is to say, an offer selling $\mathcal{A}$ for $\mathcal{B}$ with a limit price of r must trade in full if $r < (1-\mu)\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$.

Intuitively, the lower the limit price, the more an offer prefers trading to not trading.

This notion of approximation is closely related to but not exactly the same as notions of approximation used in the theoretical literature on Arrow-Debreu exchange markets (e.g., [127], Definition 1). In particular, we find it valuable in SPEEDEX to distinguish between the two types of approximation error (and measure each separately) and SPEEDEX must maintain certain guarantees exactly (e.g., assets must be conserved, and no offer can trade outside its limit price).

A.3 Tâtonnement Modifications

A.3.1 Price Update Rule

One significant algorithmic difference between the Tâtonnement implemented within SPEEDEX and the Tâtonnement described in Codenotti et al. [127] is the method in which Tâtonnement adjusts prices in response to a demand query. Codenotti et al. use an additive rule that they find amenable to theoretical analysis. If $Z(p)$ is the market demand at prices p, they update prices according to the following rule:

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}} + Z_{\mathcal{A}}(p)\delta \tag{A.1}$$

for some constant δ. The authors show that there is a sufficiently small δ so that Tâtonnement is guaranteed to move closer to an equilibrium after each step.

The relevant constant is unfortunately far too small to be usable in practice, and more generally, we want an algorithm that can quickly adapt to a wide variety of market conditions (not one that always proceeds at a slow pace).

First, we update prices multiplicatively, rather than additively. This dramatically reduces the number of required rounds, especially when Tâtonnement starts at prices that are far from the clearing prices.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1 + Z_{\mathcal{A}}(p)\delta) \tag{A.2}$$

Second, we normalize asset amounts by asset prices, so that our algorithm will be invariant to redenominating an asset. It is equivalent to trade 100 pennies or 1 USD, and our algorithm performs better when it can take that kind of context into account.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1 + p_{\mathcal{A}} Z_{\mathcal{A}}(p)\delta) \tag{A.3}$$

Next, we make δ a variable factor. We use a heuristic to guide the dynamic adjustment. Our experiments used the l^2 norm of the price-normalized demand vector, $\sum_{\mathcal{A}}(p_{\mathcal{A}} Z_{\mathcal{A}}(p))^2$; other natural heuristics (i.e. other l^p norms) perform comparably (albeit not quite as well). In every round, Tâtonnement computes this heuristic at its current set of candidate prices, and at the prices to which it would move should it take a step with the current step size. If the heuristic goes down, Tâtonnement makes the step and increases the step size, and otherwise decreases the step size. This is akin to a backtracking line search [75, 99] with a weakened termination condition.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1 + p_{\mathcal{A}} Z_{\mathcal{A}}(p)\delta_t) \tag{A.4}$$

Finally, we normalize adjustments by a trade volume factor $\nu_{\mathcal{A}}$. Without this adjustment factor, computing prices when one asset is traded much less than another asset takes a large number of rounds, simply because the lesser traded asset's price updates are always of a lower magnitude than those of the more traded asset. Many other numerical optimization problems run most quickly when gradients are normalized (e.g., see [94]).

$\nu_{\mathcal{A}}$ need not be perfectly accurate—indeed, knowing the factor exactly would require first computing clearing prices—but we can estimate it well enough from the trading volume in prior blocks and from trading volume in earlier rounds of Tâtonnement (specifically, we use the minimum of the amount of an asset sold to the auctioneer and the amount bought from the auctioneer). Real-world deployments could estimate these factors using external market data.

Putting everything together gives the following update rule:

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}} \left(1 + p_{\mathcal{A}} Z_{\mathcal{A}}(p) \delta_t \nu_{\mathcal{A}}\right) \tag{A.5}$$

The step size is represented internally as a 64-bit integer and a constant scaling factor. As mentioned in §3.5.2, we run several copies of Tâtonnement in parallel with different scaling factors and different volume normalization strategies and take whichever finishes first as the result.

Heuristic Choice

A natural question is why do we use the seemingly theoretically unfounded l^2 norm of the demand vector as our line-search heuristic. A typical line search in an optimization context uses the convex objective function of the optimization problem (e.g., [99]). Devanur et al. [146] even give a convex objective function for computing exchange market equilibria, which we reproduce below (in a simplified form):

$$\sum_{i:mp_i < \frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}} p_{\mathcal{S}_i} E_i \ln\left(mp_i \frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}\right) - y_i \ln(mp_i) \tag{A.6}$$

for mp_i the minimum limit price of an offer i that sells E_i units of good $\mathcal{S}_i$ and buys good $\mathcal{B}_i$, and $y_i = x_i p_{\mathcal{S}_i}$ for x_i the amount of $\mathcal{S}_i$ sold by the offer to the market.

This objective is accompanied by an asset conservation constraint for each asset $\mathcal{A}$:

$$\sum_{i:\mathcal{S}_i=\mathcal{A}} y_i = \sum_{i:\mathcal{B}_i=\mathcal{A}} y_i \tag{A.7}$$

However, unlike the problem formulation in [146], Tâtonnement does not have decision variables $\{y_i\}$. Rather, Tâtonnement pretends offers respond rationally to market prices, and then adjusts prices so that constraints become satisfied. As such, mapping our algorithms onto the above formulation would mean that $y_i = p_{\mathcal{S}_i} E_i$ if $mp_i < \frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}$ and 0 otherwise (although §A.3.2 would slightly change this picture). This would make the objective universally 0, and thus not useful.

We could incorporate the constraints into the objective by using the Lagrangian of the above problem, which gives the objective

$$\sum_{\mathcal{A}} \lambda_{\mathcal{A}} (\sum_{i:\mathcal{S}_i=\mathcal{A}} y_i(p) - \sum_{i:\mathcal{B}_i=\mathcal{A}} y_i(p)) \tag{A.8}$$

for a set of langrange multipliers $\{\lambda_{\mathcal{A}}\}$.

We write $y_i(p)$ to denote that in this formulation, offer behavior is directly a function of prices. It appears difficult to use equation A.8 directly as an objective to minimize, as it is nonconvex and the gradients of the functions $y_i(\cdot)$ are numerically unstable (even with the application of §A.3.2).

However, observe that equation A.8 is another way of writing "the l^1 norm of the net demand vector" (weighted by the lagrange multipliers). We use the l^2 norm instead of the l^1 to sidestep the need to actually solve for these multipliers.

An observant reader might notice that the derivative of Equation A.8 with respect to $\lambda_{\mathcal{A}}$ is the amount by which (the additive version of) Tâtonnement updates $p_{\mathcal{A}}$. This might suggest using $p_{\mathcal{A}}$ in place of $\lambda_{\mathcal{A}}$ in equation A.8. However, that search heuristic performs extremely poorly.

A.3.2 Demand Smoothing

Observe that the demand of a single offer is a (discontinuous) step function; an offer trades in full when the market exchange rate exceeds its limit price, and not at all when the market rate is less than its limit price.

These discontinuities are difficult for Tâtonnement. (Analogously, many optimization problems struggle on nondifferentiable objective functions.) As such, we approximate the behavior of each offer with a continuous function.

Recall that §A.2 measures one form of approximation error (using the parameter μ) which asks how closely realized offer behavior matches optimal offer behavior. Specifically, SPEEDEX wants to maintain the guarantee that every offer (selling $\mathcal{A}$ for $\mathcal{B}$) with a limit price below $(1-\mu)\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$ trades in full, and those with limit prices above $\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$ trade not at all.

As such, SPEEDEX has the flexibility to specify offer behavior on the gap between $(1-\mu)\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$ and $\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$. Instead of a step function, SPEEDEX linearly interpolates across the gap. That is to say, if $\alpha = \frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$, we say that an offer with limit price $(1-\mu)\alpha \leq \beta \leq \alpha$ sells an $\frac{\alpha-\beta}{\mu\alpha}$ fraction of its assets.

Observe that as μ gets increasingly small, this linear interpolation becomes an increasingly close approximation of a step function. This explains some of the behavior in Figure 3.2, particularly why the price computation problem gets increasingly difficult as μ decreases.

A.3.3 Periodic Feasibility Queries

Tâtonnement's linear interpolation simplifies computing each round, but also restricts the range of prices that meet the approximation criteria, as it does not capitalize on the flexibility we have in handling offers within μ of the market price. As a result, Tâtonnement may arrive at adequate prices without recognizing that fact. To identify good valuations, SPEEDEX runs the more expensive linear program every 1,000 iterations of Tâtonnement.

A.4 Linear Program

Recall that the role of the linear program in SPEEDEX is to compute the maximum amount of trading activity possible at a given set of prices. That is to say, Tâtonnement first computes an approximate set of market clearing prices, and then SPEEDEX runs this linear program taking the output of Tâtonnement as a set of input, constant parameters.

Throughout the following, we denote the price of an asset $\mathcal{A}$ (as output from Tâtonnement) as $p_{\mathcal{A}}$, and the amount of $\mathcal{A}$ sold in exchange for $\mathcal{B}$ as $x_{\mathcal{A},\mathcal{B}}$. We will also denote the two forms of approximation error as ε and μ, as defined in §A.2.

To maintain asset conservation, the linear program must satisfy the following constraint for every asset $\mathcal{A}$:

$$\sum_{\mathcal{B}} x_{\mathcal{A},\mathcal{B}} \geq \sum_{\mathcal{B}} (1-\varepsilon)\frac{p_{\mathcal{B}}}{p_{\mathcal{A}}} x_{\mathcal{B},\mathcal{A}}$$

Define $U_{\mathcal{A},\mathcal{B}}$ to be the upper bound on the amount of $\mathcal{A}$ that is available for sale by all offers with in the money limit prices (i.e., limit prices at or below $\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$), and define $L_{\mathcal{A},\mathcal{B}}$ to be the lower bound on the amount of $\mathcal{A}$ that must be exchanged for $\mathcal{B}$ if SPEEDEX is to be μ-approximate (i.e., execute all offers with minimum prices at or below $(1-\mu)\frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$, as described in §A.2).

Then the linear program must also satisfy the constraint, for every asset pair $(\mathcal{A}, \mathcal{B})$,

$$L_{\mathcal{A},\mathcal{B}} \leq x_{\mathcal{A},\mathcal{B}} \leq U_{\mathcal{A},\mathcal{B}}$$

Informally, the goal of our linear program is to maximize the total amount of trading activity. Any measurement of trading activity needs to be invariant to redenominating assets; intuitively, it is the same to trade 1 USD or 100 pennies. As such, the objective of our linear program is:

$$\sum_{\mathcal{A},\mathcal{B}} p_{\mathcal{A}} x_{\mathcal{A},\mathcal{B}}$$

Putting this all together gives the following linear program (let $\mathfrak{A}$ be the set of all assets):

$$\max \sum_{\mathcal{A},\mathcal{B}} p_{\mathcal{A}} x_{\mathcal{A},\mathcal{B}} \tag{A.9}$$

$$s.t.\ p_{\mathcal{A}} L_{\mathcal{A},\mathcal{B}} \le p_{\mathcal{A}} x_{\mathcal{A},\mathcal{B}} \le p_{\mathcal{A}} U_{\mathcal{A},\mathcal{B}}(p) \quad \forall \mathcal{A}, \mathcal{B} \in \mathfrak{A},\ (\mathcal{A} \neq \mathcal{B}) \tag{A.10}$$

$$p_{\mathcal{A}} \sum_{\mathcal{B} \in \mathfrak{A}} x_{\mathcal{A},\mathcal{B}} \ge (1-\varepsilon) \sum_{\mathcal{B} \in \mathfrak{A}} p_{\mathcal{B}} x_{\mathcal{B},\mathcal{A}} \quad \forall \mathcal{A} \in \mathfrak{A} \tag{A.11}$$

From the point of view of the linear program, $p_{\mathcal{A}}$ is a constant (for each asset $\mathcal{A}$). As such, this optimization problem is in fact a linear program.

It is possible that Tâtonnement could output prices where this linear program is infeasible (this is the case of the Tâtonnement timeout, as discussed in §3.6). In these cases, we set the lower bound on each $x_{\mathcal{A},\mathcal{B}}$ to be 0 instead of $L_{\mathcal{A},\mathcal{B}}$. This change makes the program always feasible (e.g., an assigment of each variable to 0 satisfies the constraints).

Observe that as written, every instance of the variable $x_{\mathcal{A},\mathcal{B}}$ appears adjacent to $p_{\mathcal{A}}$. We can simplify the program by replacing each occurrence of $p_{\mathcal{A}} x_{\mathcal{A},\mathcal{B}}$ by a new variable $y_{\mathcal{A},\mathcal{B}}$. After solving the program, we can compute $x_{\mathcal{A},\mathcal{B}}$ as $\frac{y_{\mathcal{A},\mathcal{B}}}{p_{\mathcal{A}}}$.

This substitution gives the following linear program:

$$\max \sum_{\mathcal{A},\mathcal{B}} y_{\mathcal{A},\mathcal{B}} \tag{A.12}$$

$$s.t.\ p_{\mathcal{A}} L_{\mathcal{A},\mathcal{B}} \le y_{\mathcal{A},\mathcal{B}} \le p_{\mathcal{A}} U_{\mathcal{A},\mathcal{B}}(p) \quad \forall (\mathcal{A}, \mathcal{B}),\ (\mathcal{A} \neq \mathcal{B}) \tag{A.13}$$

$$\sum_{\mathcal{B} \in \mathfrak{A}} y_{\mathcal{A},\mathcal{B}} \ge (1-\varepsilon) \sum_{\mathcal{B} \in \mathfrak{A}} y_{\mathcal{A},\mathcal{B}} \quad \forall \mathcal{A} \tag{A.14}$$

The Stellar implementation charges no transaction commission (i.e., sets ε to 0) in its SPEEDEX deployment. This makes the linear program into an instance of the maximum circulation problem (i.e., variable $y_{\mathcal{A},\mathcal{B}}$ denotes the flow from vertex $\mathcal{A}$ to vertex $\mathcal{B}$). It is well known that the constraint matrices of these problems are totally unimodular (Chapter 19, Example 4 [283]). This means that it always has an integral solution (Theorem 19.1, [283]) and can be solved by specialized algorithms (such as those outlined in [210]). Some of these algorithms run substantially faster than general simplex-based solvers.

A.5 Market Structure Decomposition

Suppose that the set of goods could be partitioned between a set of numeraires, which might be traded with any other asset, and a set of stocks, which are only traded with one of the pricing assets.

Then SPEEDEX could compute a batch equilibrium by first computing an equilibrium taking into account only trades between pricing assets, then computing an equilibrium exchange rate for every stock between the stock and its pricing asset, and finally combining the results.

More specifically:

Theorem A.5.1. *Let $\mathfrak{A}$ be the set of numeraires and $\mathfrak{S}$ the set of stocks. A stock $\mathcal{S} \in \mathfrak{S}$ is traded with asset $a(\mathcal{S}) \in \mathfrak{A}$.*

Suppose (p, x) is an equilibrium for the restricted market instance considering only the numeraires. For each $\mathcal{S} \in \mathfrak{S}$, let (r, y) be an equilibrium for the restricted market instance considering only $\mathcal{S}$ and $a(\mathcal{S})$.

Then (p', x') is an equilibrium for the entire market instance, where

1. $p'_{\mathcal{A}} = p_{\mathcal{A}}$ *for* $\mathcal{A} \in \mathfrak{A}$
2. $p'_{\mathcal{S}} = \left(r_{\mathcal{S}}/r_{a(\mathcal{S})}\right) p_{a(\mathcal{S})}$
3. $x'_{\mathcal{A},\mathcal{B}} = x_{\mathcal{A},\mathcal{B}}$ *for* $\mathcal{A}, \mathcal{B} \in \mathfrak{A}$
4. $x'_{\mathcal{S},a(\mathcal{S})} = y_{\mathcal{S},a(\mathcal{S})}$
5. $x' = 0$ *otherwise*

Proof. More generally, let G be a graph whose vertices are the traded assets and which contains an edge $(\mathcal{A}, \mathcal{B})$ if $\mathcal{A}$ and $\mathcal{B}$ can be traded directly.

Decompose G into an arbitrary set of edge-disjoint subgraphs $\{G_i\}$, such that any two subgraphs G_i, G_j share at most one common vertex. Then define a graph H whose vertices are the subgraphs G_i, and where a subgraph G_i is connected to G_j if G_i and G_j share a common vertex.

If H is acyclic, then an equilibrium can be reconstructed from equilibria computed independently on each G_i.

We reconstruct a unified set of prices iteratively, traversing along H. Given adjacent G_i and G_j sharing common vertex v_{ij}, let (p^i, x^i) and (p^j, x^j) be equilibria on G_i and G_j, respectively, rescale all of the prices p^j by $p^i_{v_{ij}}/p^j_{v_{ij}}$.

This rescaling constructs a new equilibria $(p^{j\prime}, x^j)$for G_j that agrees with that of G_i on the price of the shared good. As such, the combined system $(p^i \cup p^{j\prime}, x^i \cup x^j)$ forms an equilibrium for $G_i \cup G_j$.

This iteration is possible precisely because H is acyclic (a cycle could prevent us from finding a rescaling of some subgraph that satisfied two constraints on the prices of shared vertices).

□

A.6 Alternative Batch Solving Strategies

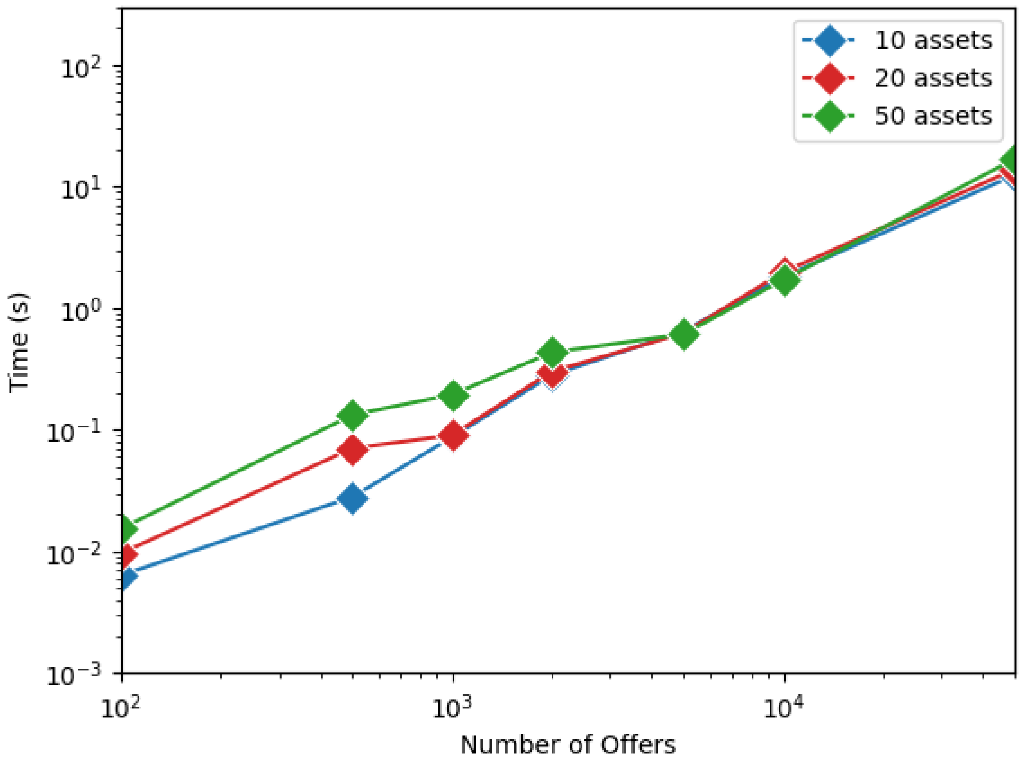


Figure A.1: Time to solve the convex program of Devanur et al. [146] using the CVXPY toolkit [149], varying the number of assets and offers.

A.6.1 Convex Optimization

We implemented the convex program of Devanur et al. [146] directly, using the CVXPY toolkit [149] backed by the ECOS convex solver [152]. Figure A.1 plots the runtimes we observed to solve the problem while varying the number of assets and offers.

The runtimes are not directly comparable to those of Tâtonnement—namely, this strategy does not have the potential to shortcircuit operation upon early arrival at an equilibrium (our notions of approximation error also do not directly translate to the notions used interally in the solver), nor is it optimized for our particular class of problems.

The important observation is that the runtime of this strategy scales linearly in the number of trade offers. Instances trading 1000 offers, for example, take roughtly 10x as long as instances trading only 100 offers.

This is not a surprising result, given that the number of variables in the convex program scales linearly with the number of trade offers.

The choice of solver strategy does not, of course, change the structure of the input problem instances. The same observation used in §3.5.1 makes it possible to refactor the convex program so that the number of variables does not depend on the number of open offers, and so that the objective (and its derivatives) can be evaluated in time logarithmic in the number of open offers.

Unfortunately, this transformation makes the objective nondifferentiable. The demand smoothing tactic of §A.3.2 gives a differentiable but not twice differentiable objective (and presents challenges regarding numerical stability of the derivative). Construction of a convex objective that approximates that of [146] while maintaining sufficient smoothness and numerical stability is an interesting open problem.

A.6.2 Mixed Integer Programming

Gnosis (Walther, [12]) give several formulations of a batch trading system as mixed-integer programming problems. These formulations track token amounts as integers (instead of as real numbers, as used in Tâtonnement's underlying mathematical formulation), which enables strict conservation of asset amounts with no rounding error.

However, mixed-integer problems appear to be computationally difficult to solve. Walther [12] finds that the runtime of this approach scales faster than linearly. Instances with more than a few hundred assets appear to be intractable for practical systems.

A.7 Tâtonnement Preprocessing

We include this section so that this paper can provide a comprehensive reference for anyone to develop their own Tâtonnement implementation.

Every demand query in Tâtonnement requires computing, for every asset pair, the amount of the asset available for sale below the queried exchange rate. As discussed in §3.9.2, Tâtonnement lays out contiguously in memory all the information it needs to return this result quickly.

For a version of Tâtonnement without the demand smoothing of §A.3.2, a demand query for exchange rate p (i.e. the ratio of the price of the sold asset to the price of the purchased asset)

$$\sum_{i:mp_i \leq p} E_i \tag{A.15}$$

where mp_i denotes the minimum price of an offer i and E_i denotes the amount of the asset offered for sale.

We can efficiently answer these queries by computing expression A.15 for every price p used as a limit price

Demand smoothing complicates the picture. The result of a demand query (with smoothing parameter μ)

$$\sum_{i:mp_i<p(1-\mu)} E_i + \sum_{i:p(1-\mu)\leq mp_i\leq p} E_i * (p - mp_i)/(p\mu) \tag{A.16}$$

We can rearrange the second term of the summation into

$$1/(p\mu) \sum_{i:p(1-\mu)\leq mp_i\leq p} (pE_i - E_i mp_i) \tag{A.17}$$

With this, we can efficiently compute the demand query after precomputing, for every unique price p that is used as a limit price, both expression A.15 and

$$\sum_{i:mp_i<p} mp_i E_i \tag{A.18}$$

The division in equation A.16 can be avoided by recognizing that Tâtonnement normalizes all asset amounts by asset valuations (so equation A.16 is always multiplied by p).

A.8 Buy Offers are PPAD-hard

A natural type of trade offer is one that offers to sell any number of units of one good to buy a fixed amount of a good (subject to some minimum price constraint). We call these *limit buy offers* (Definition 6.1.3).

These offers unfortunately do not satsify a property known as "Weak Gross Substitutability" (WGS, see e.g., [127]). This property captures the core logic of Tâtonnement. If the price of one good rises, the net demand for that good should fall, and the net demand for every other good should rise (or at least, not decrease). Limit sell offers always satisfy this property, but limit buy offers may not.

Example A.8.1. *Consider a buy offer, as in Definition 6.1.3, which offers to buy* 100 *USD selling as few EUR as possible, if and only if one EUR trades for at least* 1.1 *USD.*

The demand of this offer, when $p_{EUR} = 2$ *and* $p_{USD} = 1$, *is* $(-50\ EUR, 100\ USD)$.

If p_{USD} *rises to* 1.6, *then the demand for the offer is* $(-80\ EUR, 100\ USD)$.

Observe that the price of USD rose and the demand for EUR fell.

Informally speaking, if offers do not satisfy the core logic of Tâtonnement's price update rule, then Tâtonnement cannot handle them in a mathematically sound manner.

More formally, Chen et al. [119] show through Theorem 7 and Example 2.4 that markets consisting of collections of limit buy offers are PPAD-hard. These theorems are phrased in the language of the Arrow-Debreu exchange market model; see §A.1 for the correspondence between SPEEDEX and this model. In fact, the utility functions used in Example 2.4 to demonstrate an example "non-monotone" (i.e., defying WGS) instance are of the type that would arise by mapping limit buy offers into the Arrow-Debreu exchange market model.

A.9 Deterministic Filtering Performance

The deterministic transaction batch pruning system works by eliminating the transactions from all of the accounts that could create an unresolvable conflict. To be specific, if the sum of the amount of an asset used (either sent in a payment option or locked to create a offer) by all of an account's transactions exceeds that account's balance, then that account's transactions are removed. If an account sends two transactions with the same sequence number (both of which have valid signatures, and the sequence numbers are higher than the sequence number of the account's most recent transaction), or two transactions cancel the same offer ID, then that account's transactions are removed. If two transactions create the same account ID, then both transactions are removed.

We generated batches of 400,000 transactions from the same synthetic transaction model as in §3.7, and then duplicated 100,000 transactions at random to create a batch of 500,000. A small number of accounts (1000) send transactions with conflicting sequence numbers. We initialize the database (again, 10 million accounts) to give each account a small amount of money, and a small number (one or two hundred) of accounts attempt to overdraft.

This filtering takes 0.13s and 0.07s seconds with 24 and 48 threads, respectively (averaged over 50 trials, after a warmup), giving a 21.0× and 38.4× speedup over the serial benchmark. On a more contested benchmark, with only 10,000 accounts (almost all of which overdraft) the maximum speedup over the single threaded trial is only 5.3×, but the overall filtering runtime is still just 0.10s. Our implementation of the filtering is not heavily optimized, but in either parameter setting, the overhead is small.

A.10 Block-STM Baseline

To provide a baseline for the measurements in Fig. 3.7, we also ran Block-STM on our hardware (with hyperthreading disabled, as in [179]). Fig. A.2 displays the results.

These performance measurements are similar, quantitatively and qualitatively, to those reported in [179] (on different hardware). Note that performance appears to reach a maximum after approximately 16 to 24 threads, and, unlike SPEEDEX, does not effectively use additional hardware beyond this point, even on relatively low-contention workloads.

A.11 Additional Implementation Details

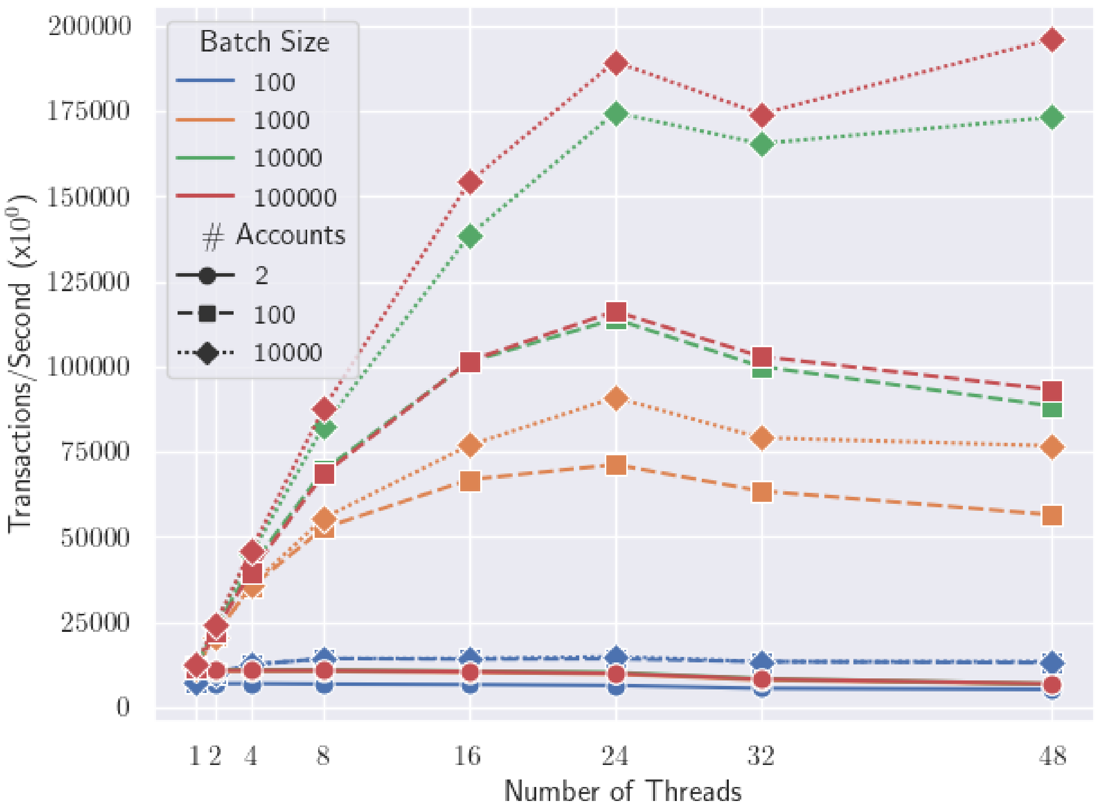


Figure A.2: Throughput of Block-STM on batches of "Aptos p2p" transactions with varying thread counts (average of 100 trials).

A.11.1 Data Organization

Account balances are stored in a Merkle-Patricia trie. However, because a trie is not self-rebalancing, its worst-case adversarial lookup performance can be slow. As such, we store account balances in memory indexed by a red-black tree, with updates pushed to the trie once per block.

For each pair of assets ($\mathcal{A}$, $\mathcal{B}$), we build a trie storing offers selling asset $\mathcal{A}$ in exchange for $\mathcal{B}$. Finally, in each block, we build a trie logging which accounts were modified.

We store information in hashable tries so that nodes can efficiently compare their database state with another replica's (to validate consensus and check for errors), and construct short proofs for users about exchange state.

A.11.2 Data Storage and Persistence

SPEEDEX uses a combination of an in-memory cache and ACID-compliant databases (several LMDB[124] instances). This choice suffices for our experiments, but a database that persists data in epochs, like Silo [296], or is otherwise optimized for batch operation might improve performance.

Our implementation uses one LMDB instance for the set of open offers, one instance for Hotstuff logs, one instance for storing block headers, and 16 instances for storing account states. LMDB is single-threaded, and we find that the throughput of one thread generating database writes does not keep up with SPEEDEX. Accounts are randomly divided between these instances, according to a hash function keyed by a (persistent) secret key (which is different per blockchain node). This key must be kept secret so as to prevent nodes from denial of service attacks.

Processing transactions in a nondeterministic order complicates recovery from a database snapshot where a block has been partially applied. Cancellation transactions, in particular, refund to an account the remainder of an offer's asset amount. We therefore cannot recover if the snapshot of the orderbooks is more recent than the snapshot of the set of account balances, and our implementation takes care to commit updates to the account LMDB instances before committing updates to the orderbook LMDB.

A.11.3 Follower Optimizations

A block proposal includes the output of Tâtonnement and the linear program in (the prices and trade amounts, as in §3.4.2). This permits the nondeterminism in Tâtonnement (§3.5.2), and lets the other nodes skip the work of running Tâtonnement.

Proposals also include, for every pair of assets, the trie key of the offer with the highest minimum price that trades in that block. When executing a proposal from another node, a follower can compare the trie key of a newly created offer with this marginal key and know immediately whether to make a trade or add the offer to the resting orderbooks. A node also defers all checks that an account balance is not overdrafted to after it has executed all the transactions in a block.

A.11.4 Replay Prevention

Transactions have per-account sequence numbers to ensure a transaction can execute only once. Many blockchains require sequence numbers from an account to increase strictly sequentially. Our implementation allows small gaps in sequence numbers, but restricts sequence numbers to increase by at most an arbitrary limit (64) in a given block. Allowing gaps simplifies some clients (such as our open-loop load generator), but more importantly lets validators efficiently track consumed sequence numbers out of order with a fixed-size bitmap and hardware atomics.

The Stellar implementation requires strictly consecutive sequence numbers, mostly for backwards compatibility.

A.11.5 Fast Offer Sorting

The running times of §3.6 do not include times to sort or preprocess offers. Naïvely sorting large lists takes a long time. Therefore, we build one trie storing offers per asset pair, and we use an offer's price, written in big-endian, as the first 6 bytes of the offer's 22-byte trie key. Constructing the trie thus automatically sorts offers by price.

Additionally, SPEEDEX executes offers with the lowest minimum prices, so a set of offers executed in a round forms a dense (set of) subtrie(s), which is trivial to remove.

A.11.6 Nondeterministic Block Assembly

As discussed in §3.3, SPEEDEX must assemble blocks of transactions in a manner that guarantees no account is overdrafted after applying all of the transactions in the block. The block proposal system (Fig. 3.1, 2) manages this by carefully controlling writes to shared state.

The proposal module takes as input a set of unconfirmed transactions (the "mempool", in typical blockchain parlance) and outputs a proposed block containing a subset of the unconfirmed transactions. For each candidate unconfirmed transaction, a thread reserves the ability to perform all necessary modifications by "locking" all relevant data elements. Once a transaction acquires all of its locks, it performs its necessary state modifications and finally releases the locks. If it cannot acquire all necessary locks, it releases any locks and excludes the transaction from the proposed block.

Conceptually, a transaction offering a trade or sending a payment must lock the number of units of assets that could be debited from the account if the operation succeeds. However, doing this with spinlocks would preclude the scalability displayed in Figure 3.7. Instead, most reservations are performed with hardware atomics to decrement the number of available units. Crediting an account can never fail because SPEEDEX caps the total amount of any asset issued at `INT64_MAX`. This process is conservative in that it may reject transactions that could have executed safely.

Unique offer IDs ensure that no offer is created twice, and atomic boolean flags ensure an offer cannot be cancelled twice. Sequence numbers can be reserved by atomic bitmaps (as in §A.11.4).

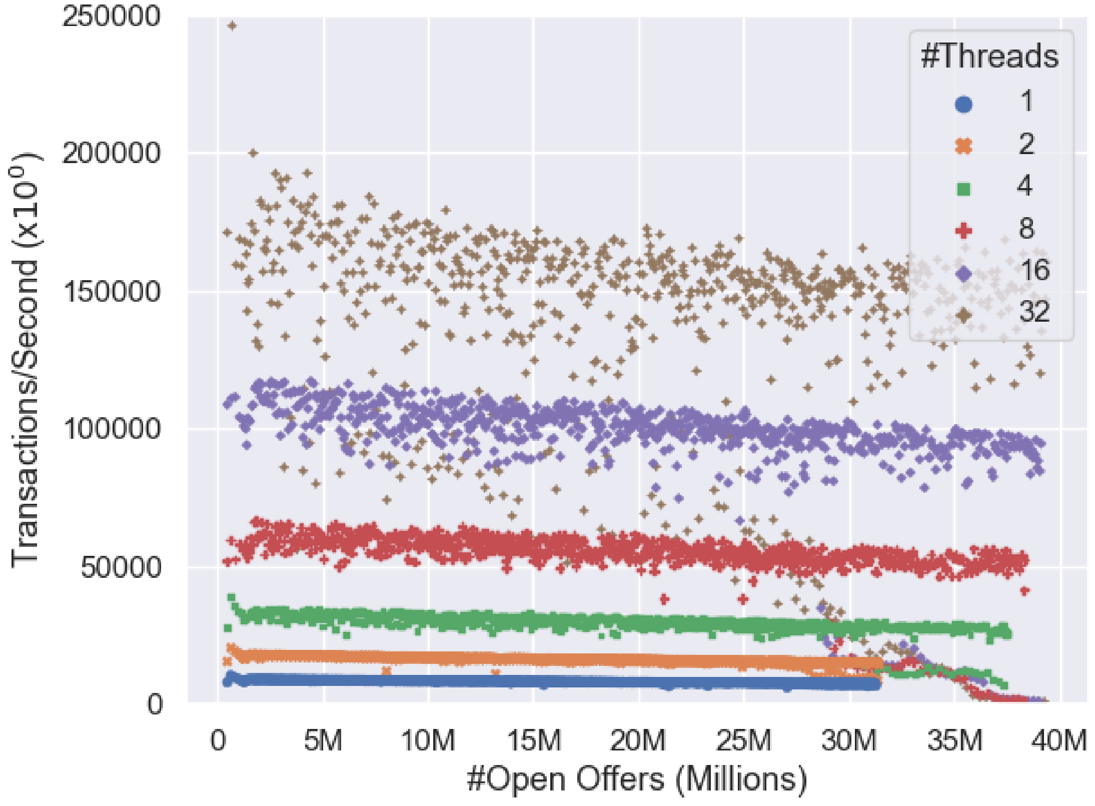


Figure A.3: Transactions per second on SPEEDEX when running with 10 replicas (on weaker hardware than in Fig. 3.3), plotted over the number of open offers.

For simplicity, our implementation does use exclusive locks when creating new accounts (which we assume occurs relatively infrequently).

A.12 Additional Replicas

SPEEDEX invokes a consensus protocol no more than once per second in our experiments. To demonstrate that this overhead is negligible, we ran SPEEDEX with 10 replicas, although with weaker hardware per replica, due to resource limitations. Each replica is one AWS c5ad.16xlarge instance, with one AMD EPYC 7R32 processor (48 CPUs @ 2.8Ghz per physical chip, 32 of which are allocated to our instances), 128 GB of memory, and two 1.1TB NVMe drives in a RAID0 configuration. Performance measurements are plotted in Figure A.3.

The overall throughput numbers are lower here than in Figure 3.3 due to the weaker hardware, but the scalability trends are the same. Doubling the thread count increases performance by a factor of between 1.8x and 1.9x, except that the jump from 16 to 32 gives a roughly 1.4x increase due to contention with background tasks (particularly logging to persistent storage).

This graph also highlights how SPEEDEX responds to insufficient hardware resources. As the number of open offers increases, SPEEDEX's memory requirements increase. Eventually, memory starts to be paged to disk, which dramatically increases disk usage and contends with the logging to persistent storage. SPEEDEX slows down in response, to ensure for safety that data in peristent storage is never too far out of sync.

Appendix B

Chapter 4 Supporting Information

B.1 Truncation of Ranked Pairs

Ranked Pairs preserves its output ordering when restricted to considering only an initial segment of its output. In the statement below, $\mathcal{A}(\cdots)$ denotes Ranked Pairs Voting (Algorithm 1).

Lemma B.1.1. *Suppose that $(\sigma_1, \ldots, \sigma_n)$ is a set of ordering votes on a set V of transactions, and let $\sigma = \mathcal{A}(\sigma_1, \ldots, \sigma_n)$.*

Consider a set $V' \subset V$ that forms the transactions in an initial segment of σ, and for each $i \in [n]$, let σ_i' be σ_i restricted to V'.

Then σ extends $\mathcal{A}(\sigma_1', \ldots, \sigma_n')$.

Proof. The proof follows by executing $\mathcal{A}(\sigma_1, \ldots, \sigma_n)$ and comparing against an execution on the restricted input.

Whenever the algorithm includes an edge, inclusion cannot create a cycle in the graph of previously included edges, so it cannot create a cycle in the graph of previously included edges restricted to V'.

Whenever the algorithm rejects an edge (tx, tx'), it must have included already a directed path from tx' to tx. If tx and tx' both lie within V' (that is, the algorithm run on the restricted input would iterate over this edge), then the previously included path must be entirely contained within V'. Otherwise, there would be some tx^* on this path in $V \setminus V'$, but that would imply that $tx*$ would come before tx in σ and thus that V' is not the set of transactions in an initial segment of σ (as $tx \in V'$). $\square$

B.2 On the Difficulty of Statically Choosing γ

One challenge with prior work on this topic is that prior systems require choosing a static value of γ [209, 208, 110]. It is not immediately obvious whether a higher or lower value of γ provides a stronger fairness guarantee. In fact, which value of γ gives the stronger guarantee depends on the scenario, as we show by example here. These examples assume that there are no faulty replicas, and therefore invoke not Definition 4.4.1 of prior work but the stronger Definition 4.4.3, which fixes the problems around batch minimality discussed in §4.4.2.

For simplicity, in this section, suppose that there are $n = 10$ replicas.

Example B.2.1 (High γ). *Suppose that* 3 *replicas receive* $tx_1 \preccurlyeq tx_2 \preccurlyeq tx \preccurlyeq tx'$*,* 3 *replicas receive* $tx_2 \preccurlyeq tx \preccurlyeq tx' \preccurlyeq tx_1$*, and* 4 *replicas receive* $tx \preccurlyeq tx' \preccurlyeq tx_1 \preccurlyeq tx_2$*.*

Then Definition 4.4.3 only ensures that $tx \preccurlyeq tx'$ *(an ordering relation on which every replica agrees) if* $\gamma \geq 0.6$*.*

Example B.2.2 (Low γ). *Suppose that* 6 *replicas receive* $tx \preccurlyeq tx'$*, and* 4 $tx' \preccurlyeq tx$*. Then Definition 4.4.3 only ensures that* $tx \preccurlyeq tx'$ *in an algorithm's output if* $\gamma \leq 0.6$*.*

Example B.2.3 (Intermediate γ). *Suppose that* 4 *replicas receive* $tx_1 \preccurlyeq tx_2 \preccurlyeq tx_3$*,* 4 *replicas receive* $tx_3 \preccurlyeq tx_1 \preccurlyeq tx_2$*, and* 2 *replicas receive* $tx_2 \preccurlyeq tx_3 \preccurlyeq tx_1$*.*

Then 80% *receive* $tx_1 \preccurlyeq tx_2$*,* 60% *receive* $tx_2 \preccurlyeq tx_3$*, and* 60% *receive* $tx_3 \preccurlyeq tx_1$*. Thus, Definition 4.4.3 only provides any constraint on the output ordering (in this case, that the strongest ordering dependency, between* tx_1 *and* tx_2*, is respected) if* $0.6 < \gamma \leq 0.8$*.*

B.3 Themis and Early Finalization

Themis [208], by design, achieves bounded liveness by acknowledging the possibility that a Condorcet cycle could be output spread across multiple rounds of its protocol.

For simplicity, assume for the example that the number of faulty replicas f is 0, and assume that γn and $\frac{n}{3}$ are integers. For simplicity of exposition, we assume that $\frac{2}{3} + \frac{1}{n} < \gamma < 1$. Analogous examples will work for other values of n and γ.

There will be five transactions in this example. Ultimately, replicas will receive transactions in the following order.

Class 1	1 replica	$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_4 \preccurlyeq tx_3 \preccurlyeq tx_5$
Class 2	$(1-\gamma)n$ replicas	$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_5 \preccurlyeq tx_4 \preccurlyeq tx_3$
Class 3	$n(\gamma - \frac{2}{3}) - 1$ replicas	$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_3 \preccurlyeq tx_5 \preccurlyeq tx_4$
Class 4	$\frac{n}{3}$ replicas	$tx_3 \preccurlyeq tx_1 \preccurlyeq tx_2 \preccurlyeq tx_5 \preccurlyeq tx_4$
Class 5	$\frac{n}{3}$ replicas	$tx_2 \preccurlyeq tx_3 \preccurlyeq tx_1 \preccurlyeq tx_5 \preccurlyeq tx_4$

Clearly, if Ranked Pairs is run on the full ordering votes, the result would have $tx_5 \preccurlyeq tx_4$, as all but 1 replica receive those transactions in that order (and that is the first edge considered by Ranked Pairs).

Themis operates in rounds, each of which has three phases. First, in the "FairPropose" phase, replicas submit ordering votes to a leader, who gathers them and publishes a proposal. Suppose that, due to the timing of messages over the network, the leader receives the following set of votes.

Class 1	$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_4 \preccurlyeq tx_3$
Class 2	$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_5 \preccurlyeq tx_4 \preccurlyeq tx_3$
Class 3	$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_3$
Class 4	$tx_3 \preccurlyeq tx_1 \preccurlyeq tx_2$
Class 5	$tx_2 \preccurlyeq tx_3 \preccurlyeq tx_1$

Themis then considers transactions by the number of replicas which include the transactions in their votes. Those in n votes are "solid", those with fewer than n but at least $\gamma n + 1$ are "shaded," and the rest are "blank." Clearly transactions tx_1, tx_2, and tx_3 are solid, tx_4 is shaded, and tx_5 is blank.

Themis then builds a dependency graph between the solid and shaded vertices. A dependency is added between two transactions tx and tx' if at least $(1-\gamma)n+1$ replicas vote $tx \preccurlyeq tx'$ and fewer replicas vote $tx' \preccurlyeq tx$.

As such, the dependencies that get added here are $tx_1 \preccurlyeq tx_2$, $tx_2 \preccurlyeq tx_3$, $tx_3 \preccurlyeq tx_1$, $tx_1 \preccurlyeq tx_4$, $tx_2 \preccurlyeq tx_4$ and $tx_4 \preccurlyeq tx_3$.

The next step is to compute the strongly connected components of this graph and topologically sort it. In this case, the graph has one component. This component is the last in the ordering that contains a solid vertex, so, per Themis's rules, this graph is the output of FairPropose.

Next, replicas send "updates" to a "FairUpdate" algorithm, which orders shaded transactions in the output of FairPropose relative to the other solid and shaded transactions in that output. This is where applying Ranked Pairs (or any ranking algorithm) to each output batch of Themis individually breaks down. The response of honest replicas at this point (namely, those in classes 3, 4 and 5) is to vote that they received tx_4 after each of tx_1, tx_2 and tx_3. In fact, no edges get added to the dependency graph at this stage, because the graph is already complete.

Finally, in "FairFinalize," Themis computes some ordering on transactions tx_1, tx_2, tx_3, and tx_4, and includes them in the output.

Observe that tx_5 is forced to come after tx_4.

B.4 Ranked Pairs and Interleaved Components

The following example demonstrates the need in Definition 4.4.5 to allow what would be distinct but incomparable batches in prior work (specifically Aequitas [209]) to be output in an interleaved ordering, instead of requiring that one be fully output before the other. As a corollary, this demonstrates that no process for ordering transactions within Aequitas's batches can always achieve the same ordering as output in Ranked Pairs.

Consider the following set of ordering votes on 8 transactions from 16 replicas. In this example, we assume no replicas are faulty.

The general pattern is two condorcet cycles of four transactions each, with one set on the extremes and one set in the middle. Differences from this pattern are bolded for clarify.

$$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_3 \preccurlyeq tx_4$$
$$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_3 \preccurlyeq tx_4$$
$$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_3 \preccurlyeq tx_4$$
$$tx_1 \preccurlyeq tx_2 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_3 \preccurlyeq tx_4$$
$$tx_2 \preccurlyeq tx_3 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq \mathbf{tx_4} \preccurlyeq \mathbf{tx_8} \preccurlyeq tx_1$$
$$tx_2 \preccurlyeq tx_3 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_4 \preccurlyeq tx_1$$
$$tx_2 \preccurlyeq tx_3 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_4 \preccurlyeq tx_1$$
$$tx_2 \preccurlyeq tx_3 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq \mathbf{tx_1} \preccurlyeq \mathbf{tx_4}$$
$$tx_3 \preccurlyeq tx_4 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_1 \preccurlyeq tx_2$$
$$tx_3 \preccurlyeq tx_4 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq \mathbf{tx_5} \preccurlyeq \mathbf{tx_8} \preccurlyeq tx_1 \preccurlyeq tx_2$$
$$tx_3 \preccurlyeq tx_4 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_1 \preccurlyeq tx_2$$
$$tx_3 \preccurlyeq tx_4 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_1 \preccurlyeq tx_2$$
$$tx_4 \preccurlyeq \mathbf{tx_5} \preccurlyeq \mathbf{tx_1} \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_2 \preccurlyeq tx_3$$
$$tx_4 \preccurlyeq tx_1 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_2 \preccurlyeq tx_3$$
$$tx_4 \preccurlyeq tx_1 \preccurlyeq tx_7 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_2 \preccurlyeq tx_3$$
$$tx_4 \preccurlyeq tx_1 \preccurlyeq tx_8 \preccurlyeq tx_5 \preccurlyeq tx_6 \preccurlyeq tx_7 \preccurlyeq tx_2 \preccurlyeq tx_3$$

Sorting edges by weight gives the following:

12/16	$tx_1 \preccurlyeq tx_2$, $tx_2 \preccurlyeq tx_3$, $tx_3 \preccurlyeq tx_4$, $tx_5 \preccurlyeq tx_6$, $tx_6 \preccurlyeq tx_7$, $tx_7 \preccurlyeq tx_8$
11/16	$tx_4 \preccurlyeq tx_1$, $tx_8 \preccurlyeq tx_5$
9/16	$tx_5 \preccurlyeq tx_1$, $tx_4 \preccurlyeq tx_8$

All others have weight 8/16 or less.

As such, when Aequitas is run with γ between 9/16 and 11/16, would recognize the sets $\{tx_1, tx_2, tx_3, tx_4\}$ and $\{tx_5, tx_6, tx_7, tx_8\}$ as incomparable batches, and would output all of one before beginning to output the other—no matter how it orders transactions within these two batches.

However, Ranked Pairs, on this input, would first select all of the weight 12/16 edges, reject the weight 11/16 edges, and accept the weight 9/16 edges. This means that, no matter how the rest of the edges are considered, Ranked Pairs will at least output $tx_5 \preccurlyeq tx_1$, $tx_1 \preccurlyeq tx_4$, and $tx_4 \preccurlyeq tx_8$.

This ordering thus cannot be the result of any Aequitas invocation with this choice of γ.

B.5 Algorithm 2 Is Not Asymptotically Live

Consider the following setting. There are two replicas (neither of which is faulty), and a countably infinite set of transactions $\{tx_1, tx_2, tx_3, \ldots\}$.

Example B.5.1.
Replica 1 receives transactions in the order
$tx_2 \preccurlyeq tx_1 \preccurlyeq tx_4 \preccurlyeq tx_3 \preccurlyeq tx_6 \preccurlyeq tx_5 \preccurlyeq \ldots \preccurlyeq tx_{2*i} \preccurlyeq tx_{2*i-1} \preccurlyeq \ldots$*, for all* $i \geq 1$*.*
Replica 2 receives transactions in the order
$tx_1 \preccurlyeq tx_3 \preccurlyeq tx_2 \preccurlyeq tx_5 \preccurlyeq tx_4 \preccurlyeq tx_7 \preccurlyeq tx_6 \ldots \preccurlyeq tx_{2*i+1} \preccurlyeq tx_{2*i} \ldots$*, for all* $i \geq 1$*.*

This means that any edge (tx_i, tx_j) *has weight* $\frac{1}{2}$ *if* $i = j+1$ *or* $j = i+1$*, and otherwise* 1 *(resp.* 0*) if* $i < j$ *(resp.* $i > j$*).*

Additionally, suppose that of the edges of weight $1/2$ *of the form* $e_i = (tx_{i+1}, tx_i)$*, the deterministic tiebreaking ordering sorts* (tx_{i+1}, tx_i) *just before* (tx_i, tx_{i+1}) *and* e_i *before* e_j *for* $i > j$*.*

Theorem B.5.2. *Algorithm 2 never outputs any transaction on any finite subset of the input in Example B.5.1.*

Proof. On any finite truncation of this input, suppose that N is the highest index such that tx_N and all tx_i for $i \leq N$ appear in the votes of both replicas.

Then tx_{N+1} must be present in the output of one of the replicas, but not both. This means that the other replica's vote (that does not contain tx_{N+1}) may contain tx_{N+2}, but the transaction after this in the replica's true vote is tx_{N+1}, so the other replica's vote must stop here. As such, the streamed ordering graph used in Algorithm 2 will contain all transactions up to tx_N, possibly tx_{N+2}, and $\hat{v}$. There will be one edge from $\hat{v}$ to tx_N, and if it is present, an edge from $\hat{v}$ to tx_{N+2}.

Algorithm 2 will accept all of the edges of weight 1 and then proceed to the edges of weight $\frac{1}{2}$. The first to be considered will be e_N, which will be marked *indeterminate* because of the path $(tx_{N-1}, \hat{v}), (\hat{v}, tx_N)$ (and so will the reversal of e_N also be marked *indeterminate*). Continuing through the sequence of edges, If e_k is marked *indeterminate*, then so too will e_{k-1}, because of the path $(tx_{k-2}, tx_k), e_k$. Note that this is the only path that Algorithm 2 considers when visiting e_{k-1}, as the only other transactions in $U_{tx_{k-1}, tx_{k-2}}$ are tx_k and tx_{k-3}.

Thus, Algorithm 2 never outputs any transactions on a finite subset of the input in Example B.5.1. □

We remark that this construction requires that the ordering between edges of weight $\frac{1}{2}$ is not isomorphic to a subset of ω.

B.6 Oracle Implementation

Algorithm 3 relies on an oracle to a past invocation of the algorithm. Implementing the oracle naively would require maintaining a map from the set of all edges of the past invocation to their status, the cost of which would grow logarithmically. However, the only edges for which a path search must be performed are those (tx, tx') for which neither tx nor tx' were included in the prior output (if either tx or tx' were in the prior output, then monotonicity immediately implies whether the edge is included or excluded). As such, the map in the oracle could be pruned of edges connected to transactions in the prior output. Although the number of such edges is not bounded, it is not guaranteed to grow over time without bound.

Appendix C

Chapter 5 Supporting Information

C.1 CFMMs and Market Scoring Rules

We highlight here for completeness the equivalence between market scoring rules [190] and CFMMs. Chen and Pennock [121] show that every prediction market, based on a market scoring rule, can be represented using some "cost function."

A prediction market trades n types of shares, each of which pays out 1 unit of a numeraire if a particular future event occurs. The cost function $C(q)$ of [121] is a map from the total number of issued shares of each event, $q \in \mathbb{R}^n$, to some number of units of the numeraire. To make a trade $\delta \in \mathbb{R}^n$ with the prediction market (i.e. to change the total number of issued shares to $q+\delta$), a user pays $z = C(q+\delta) - C(q)$ units of the numeraire to the market.

One discrepancy is that traditional formulations of prediction markets (e.g. [121, 191]) allow an arbitrary number of shares to be issued by the market maker, but the CFMMs described in this work trade in assets with finite supplies. Suppose for the moment, however, that a CFMM could possess a negative quantity of shares (with the trading function f defined on the entirety of $\mathbb{R}^n$, instead of just the positive orthant). This formulation of a prediction market directly gives a CFMM that trades the n shares and the numeraire, with trading function $f(r,z) = -C(-r) + z$ for $r \in \mathbb{R}^n$ the number of shares owned by the CFMM, and z the number of units of the numeraire owned by the CFMM. Observe that for any trade δ and $dz = C(-(r+\delta)) - C(-r)$, $f(r,z) = f(r+\delta, z+dz)$. This establishes the correspondence between prediction markets and CFMMs.

In our examples with the LMSR, we consider a CFMM for which $z = 0$ (i.e., it doesn't exchange shares for dollars, but only shares of one future event for shares of another future event). The cost function $C(r)$ for the LMSR is $\log(\sum_{i=1}^n \exp(-r_i))$. The CFMM representation with this cost function follows by setting it to a constant.

C.2 Continuous Trade Size Distributions

Definition C.2.1. *Let size$(\cdot)$ be some distribution on $\mathbb{R}_{\geq 0}$ with support in a neighborhood of* 0*.*

A trader appears at every timestep. The trade has size k units of Y, where k is drawn from size$(\cdot)$. A trade buys or sells from the CFMM with equal probability.

This definition implicitly encodes an assumption that the amount of trading from X to Y is balanced in expectation against the amount of trading from Y to X.

An additional assumption makes this setting analytically tractable.

Assumption C.2.2 (Strict Slippage)**.** *Trade requests measure slippage relative to the post-trade* spot exchange rate *of the CFMM, not the overall exchange rate of the trade.*

In other words, a trade request succeeds if and only if it would move the CFMM's reserves to some state within $L_\varepsilon(\hat{p})$.

We now analyze the Markov chain over the CFMM's state, the stationary distribution of which gives us the trade failure probability under Assumption C.2.2.

Lemma C.2.3. *Let M be the Markov chain defined by the state of Y in the asset reserves of the CFMM with $\mathcal{Y} \in L_\varepsilon(\hat{p})$ and transitions induced by trades drawn from the distribution in Definition C.2.1. Under Assumption C.2.2, the stationary distribution of M is uniform over $L_\varepsilon(\hat{p})$.*

Proof. Let $\mu(\cdot)$ be the uniform measure on $L_\varepsilon(\hat{p})$ and let $\tau(v, A)$ be the state transition kernel induced by the trade distribution. Specifically, given the trade size distribution $size(v)$ on the probability that the CFMM sells (for $v > 0$) or buys (for $v < 0$) $|v|$ units of Y, $\tau(v, A)$ measures the probability that for any set A, the CFMM is in a state in A after attempting a trade of size v. It suffices to show that μ is an invariant measure of the Markov chain that is induced on $L_\varepsilon(\hat{(p})$, and that this invariant measure is unique.

Note that from any $y \in L_\varepsilon(\hat{p})$, after a trade of size v, the Markov chain lands in a set A if either $y + v \in A$ (that is, the trade succeeds) or if $y \in A$ and $y + v \notin L_\varepsilon(\hat{p})$ (that is, the trade fails and the initial state was in A). Note that trade success and failure are mutually exclusive events.

$$
\begin{aligned}
&\int\limits_{L_\varepsilon(\hat{p})} \mu(y)\tau(y, A)dy \\
&= \int\limits_{L_\varepsilon(\hat{p})} \int\limits_{-\infty}^{\infty} size(v)(\mathbb{1}(y + v \in A) + \mathbb{1}(y \in A \wedge y + v \notin L_\varepsilon(\hat{p})))dv\ dy \\
&= \int\limits_{A} \int\limits_{-\infty}^{\infty} size(v)(\mathbb{1}(y + v \in L_\varepsilon(\hat{p})))dv\ dy + \int\limits_{A} \int\limits_{-\infty}^{\infty} \mathbb{1}(y + v \notin L_\varepsilon(\hat{p})))dv\ dy \\
&= \mu(A)
\end{aligned}
$$

where the second equality follows from the symmetricity of the trade size distribution (as in Definition C.2.1).

Because the trade size distribution is supported on a neighborhood of 0 (and $L_\varepsilon(\hat{p})$ is a connected interval), for any set A of nonzero measure, the probability of a transition from $L_\varepsilon(\hat{p}) \setminus A$ to A is nonzero, so the Markov chain must A infinitely many times. As such, $\mu(\cdot)$ is the unique invariant measure on $L_\varepsilon(\hat{p})$ (by Theorem 1 of [192]).

□

Proposition C.2.4. *The probability that a trade of size k units of Y fails is approximately* $\min(1, \frac{k}{|L_\varepsilon(\hat{p})|})$, *where the approximation error is up to Assumption C.2.2.*

Proof. The probability that a (without loss of generality) sell of size k units of Y fails is equal to the probability that a state y, drawn uniformly from the range $L_\varepsilon(p) = [y_1, y_2]$, lies in the range $[y_2 - k, y_2]$. Lemma C.2.3 shows this probability is $\min(1, \frac{k}{y_2 - y_1})$. □

C.3 Omitted Proofs

C.3.1 Omitted Proofs of §5.2 and §5.3

Restatement (Observation 5.2.4)**.** *If f is strictly quasi-concave and differentiable, then for any constant K and spot exchange rate p, the point (x, y) where $f(x, y) = K$ and p is a spot exchange rate at (x, y) is unique.*

Proof. A constant $K = f(X_0, Y_0)$ defines a set $\{x : f(x) \geq K\}$. Because f is strictly quasi-concave, this set is strictly convex. Trades against the CFMM (starting from initial reserves (X_0, Y_0)) move along the boundaries of this set. Because this set is strictly convex, no two points on the boundary can share a gradient (or subgradient). □

Restatement (Observation 5.2.5)**.** *If f is strictly increasing in both X and Y at every point on the positive orthant, then for a given constant function value K, the amount of Y in the CFMM reserves uniquely specifies the amount of X in the reserves, and vice versa.*

Proof. If not, then f would be constant on some line with either X or Y constant. □

Restatement (Observation 5.2.6)**.** *$\mathcal{Y}(p)$ is monotone nondecreasing.*

Proof. If $\mathcal{Y}(p)$ is decreasing, the level set of f, i.e., $\{(x, y) : f(x, y) \geq K\}$ cannot be convex. □

Restatement (Lemma 5.3.8)**.** *The function $\mathcal{Y}(\cdot)$ is differentiable when the trading function f is twice-differentiable on the nonnegative orthant, f is 0 when $x = 0$ or $y = 0$, and Assumption 5.2.2 holds.*

Proof. Observation 5.2.5 implies that the amount of Y in the reserves can be represented as a function $\hat{\mathcal{Y}}(x)$ of the amount of X in the reserves. By assumption, the level sets of f (other than for $f(\cdot) = 0$) cannot touch the boundary of the nonnegative orthant.

Because f is differentiable and increasing at every point in the positive orthant, the map $g(x)$ from reserves x to spot exchange rates at $(x, \hat{\mathcal{Y}}(x))$ must be a bijection from $(0, \infty)$ to $(0, \infty)$. Because f is twice-differentiable, $g(x)$ must be differentiable, and so the map $h(p) = g^{-1}(p)$ must also be differentiable. The map $\mathcal{Y}(p)$ from spot exchange rates to reserves Y is equal to $\hat{\mathcal{Y}}(h(p))$, and so $\hat{Y}(p)$ is differentiable because $\hat{\mathcal{Y}}(\cdot)$ is differentiable and $h(\cdot)$ is differentiable. □

Restatement (Lemma 5.3.9)**.** *If the function $\mathcal{Y}(\cdot)$ is differentiable, then $L(p) = \frac{d\mathcal{Y}(p)}{d\ln(p)}$.*

Proof. Follows from Definitions 5.3.6 and 5.3.7. □

C.3.2 Omitted Proof of Corollary 5.4.6

Restatement (Corollary 5.4.6)**.** *Any two beliefs ψ_1, ψ_2 give the same optimal liquidity allocations if there exists a constant $\alpha > 0$ such that for every θ,*

$$\int_r \psi_1(r\cos(\theta), r\sin(\theta))dr = \alpha \int_r \psi_2(r\cos(\theta), r\sin(\theta))dr$$

Proof. Follows by substitution. α rescales the derivative of the objective with respect to every variable by the same constant, and thus does not affect whether an allocation is optimal. □

C.3.3 Omitted Proof of Corollary 5.4.7

Restatement (Corollary 5.4.7)**.** *Define $\varphi_\psi(\theta) = \int_r \psi(r\cos(\theta), r\sin(\theta))dr$. Then*

$$\iint_{p_X, p_Y} \frac{\psi(p_X, p_Y)}{p_Y L(p_X/p_Y)} dp_X \ dp_Y = \int_p \frac{\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}{L(p)} dp$$

Proof.

$$\begin{aligned}\int_{p_X, p_Y} \frac{\psi(p_X, p_Y)}{p_Y L(p_X/p_Y)} dp_X \ dp_Y &= \int_\theta \frac{\varphi_\psi(\theta)}{L(\cot(\theta))\sin(\theta)} d\theta \\ &= \int_p \frac{\varphi_\psi(\theta)\sin^2(\theta)}{L(\cot(\theta))\sin(\theta)} dp \\ &= \int_p \frac{\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}{L(p)} dp\end{aligned}$$

The first line follows by Lemma 5.4.5 (recall that $dp_X \ dp_Y = r \ dr \ d\theta$), the second by substitution of $p = \cot(\theta)$ and $d\theta = -\sin^2(\theta)dp$ (and changing the direction of integration — recall $\theta = 0$ when $p = \infty$), and the third by substitution. □

C.3.4 Omitted Proof of Lemma 5.4.8

Restatement (Lemma 5.4.8)**.** *The optimization problem of Theorem 5.4.4 always has a solution with finite objective value.*

Proof. Set $L(p) = 1$ for $p \leq 1$ and $L(p) = \varphi_\psi(\cot^{-1}(p))/p^2$ otherwise. Then

$$\begin{aligned}
&\int_p \frac{\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}{L(p)}dp \\
&\leq \int_p \frac{\varphi_\psi(\cot^{-1}(p))}{L(p)}dp \\
&\leq \int_0^1 \varphi_\psi(\cot^{-1}(p))dp + \int_1^\infty \frac{dp}{p^2}
\end{aligned}$$

The first term of the last line is finite, as per our assumption on trader beliefs.

Set $Y_0 = \int_0^{p_0} \frac{L(p)dp}{p}$ and $X_0 = \int_{p_0}^\infty \frac{L(p)dp}{p^2}$. Clearly both X_0 and Y_0 are finite. Finally, rescale each $L(p)$, X_0, and Y_0 by a factor of $\frac{B}{P_X X_0 + P_Y Y_0}$ to get a new allocation $L'(p)$, X_0', and Y_0' satisfing the constraints and that still gives a finite objective value. □

C.3.5 Omitted Proof of Lemma 5.4.10

Restatement. *The following hold at any optimal solution.*

1. $\int_0^{p_0} \frac{L(p)}{p}dp = Y_0$
2. $\int_{p_0}^\infty \frac{L(p)}{p^2}dp = X_0$
3. $X_0 P_X + Y_0 P_Y = B$

Proof. The third equation holds since the objective function is strictly decreasing in at least one $L(p)$ (where the belief puts a nonzero probability on the exchange rate p), so any unallocated capital could be allocated to increase this $L(\cdot)$ on a neighborhood of $L(p)$ and reduce the objective.

The first equation holds because any unallocated units of X could be allocated to $L(p')$ for a set of p' in a neighborhood of some $p \leq p_0$ and thereby reduce the objective. If there is no $p \leq p_0$ where the belief puts a nonzero probability, then all of the capital allocated by the third constraint to X_0 could be reallocated into increasing Y_0 and thereby decreasing the objective.

The second equation follows by symmetry with the argument for the first. □

C.3.6 Omitted Proof of Corollary 5.4.12

Restatement (Corollary 5.4.12)**.** *The integral $\mathcal{Y}(\tilde{p}) = \int_0^{\tilde{p}} \frac{L(p)dp}{p}$ is well defined for every $\tilde{p}$ and $\mathcal{Y}(\cdot)$ is monotone nondecreasing and continuous.*

Proof. The last item of Lemma 5.4.10 shows that $L(p) \neq 0$ if and only if $\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p)) \neq 0$. When $L(p)$ is nonzero, it is either $\sqrt{\frac{p^2}{\lambda_X}\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}$ or $\sqrt{\frac{p}{\lambda_Y}\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}$ (depending on the value of p).

By our assumption on trader beliefs, $\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))$ is a well-defined function of p and is integrable. Thus, both $\sqrt{\frac{p^2}{\lambda_X}\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}$ and $\sqrt{\frac{p}{\lambda_Y}\varphi_\psi(\cot^{-1}(p))\sin(\cot^{-1}(p))}$ are integrable. Monotonicity follows from $L(p) \geq 0$ and continuity from basic facts about integrals. □

C.3.7 Omitted Proof of Corollary 5.4.15

Restatement (Corollary 5.4.15)**.** *A liquidity allocation $L(\cdot)$ and an initial spot exchange rate p_0 are sufficient to uniquely specify an equivalence class of beliefs (as defined in Corollary 5.4.6) for which $L(\cdot)$ is optimal.*

Proof. It suffices to uniquely identify $\varphi_\psi(\cot^{-1}(p))$ for each p, up to some scalar. Lemma 5.4.13 shows that $\varphi_\psi(\cot^{-1}(p))$ is a function of an optimal $L(p)$ and Lagrange multipliers λ_X or λ_Y, and because $\lambda_X\frac{P_X}{P_Y} = \lambda_Y$, we must have that φ_ψ is specified by $L(p)$ and p_0 up to some scalar λ_X. □

C.3.8 Omitted Proof of Corollary 5.4.16

Restatement (Corollary 5.4.16)**.** *Let P_X and P_Y be initial reference valuations, and let $L(\cdot)$ denote a liquidity allocation. Define the belief $\psi(p_X, p_Y)$ to be $\frac{(L(p_X/p_Y))^2}{p_X/p_Y}$ when $p_X \in (0, P_X]$ and $p_Y \in (0, P_Y]$, and to be 0 otherwise. Then $L(\cdot)$ is the optimal allocation for $\psi(\cdot,\cdot)$.*

Proof. Recall the definition of $\varphi_\psi(\cdot)$ in 5.4.7. For the given belief function ψ, standard trigonometric arguments show that when $p \geq p_0$, we have $\varphi_\psi(\cot^{-1}(p)) = \frac{P_X L(p)^2/p}{\cos(\cot^{-1}(p))}$ and that when $p \leq p_0$, we have $\varphi_\psi(\cot^{-1}(p)) = \frac{P_Y L(p)^2/p}{\sin(\cot^{-1}(p))}$.

Let $\hat{L}(p)$ be the allocation that results from solving the optimization problem for minimising the expected CFMM inefficiency for belief ψ. Lemma 5.4.13 part 3, gives the complementary slackness condition of $\hat{L}(p)$ and its corresponding Lagrange multiplier. With this, Lemma 5.4.11 gives the following: when $p \geq p_0$, $\frac{\lambda_B P_X}{p^2} = \frac{1}{\hat{L}(p)^2}(P_X L(p)^2/p)/p$, and when $p \leq p_0$, $\frac{\lambda_B P_Y}{p} = \frac{1}{\hat{L}(p)^2}(P_Y L(p)^2/p)$.

In other words, for all p, $\lambda_B = \frac{L(p)^2}{\hat{L}(p)^2}$, so $L(\cdot)$ and $\hat{L}(\cdot)$ differ by at most a constant multiplicative factor. But both allocations use the same budget, so it must be that $\lambda_B = 1$ and $\hat{L}(\cdot) = L(\cdot)$. □

C.3.9 Omitted Proof of Corollary 5.4.17

Restatement (Corollary 5.4.17)**.** *Let ψ_1, ψ_2 be any two belief functions (that give φ_{ψ_1} and φ_{ψ_2}) with optimal allocations $L_1(\cdot)$ and $L_2(\cdot)$, and let $L(\cdot)$ be the optimal allocation for $\psi_1 + \psi_2$. Then $L^2(\cdot)$ is a linear combination of $L_1^2(\cdot)$ and $L_2^2(\cdot)$.*

Further, when φ_{ψ_1} and φ_{ψ_2} have disjoint support, $L(\cdot)$ is a linear combination of $L_1(\cdot)$ and $L_2(\cdot)$.

Proof. Note that $\int_r \psi(r\cos(\theta), r\sin(\theta)dr$ is a linear function of each $\psi(p_X, p_Y)$, and thus $\varphi_{\psi_1+\psi_2}(\cdot) = \varphi_{\psi_1}(\cdot) + \varphi_{\psi_2}(\cdot)$. For any p with $p \geq p_0$ and nonzero $\varphi_{\psi_1}(\cot^{-1}(p))$, we must have that $L_1(p)^2 = \frac{p^2}{\lambda_{1,X}}\varphi_{\psi_1}(\cot^{-1}(p))\sin(\cot^{-1}(p))$.

Similarly, for nonzero $\varphi_{\psi_2}(\cot^{-1}(p))$, $L_2(p)^2 = \frac{p^2}{\lambda_{2,X}}\varphi_{\psi_2}(\cot^{-1}(p))\sin(\cot^{-1}(p))$.

If either φ_{ψ_1} or φ_{ψ_2} is nonzero at $\cot^{-1}(p)$, then

$$L(p)^2 = \frac{p^2}{\lambda_X}(\varphi_{\psi_1}(\cot^{-1}(p))\sin(\cot^{-1}(p)) + \varphi_{\psi_2}(\cot^{-1}(p))\sin(\cot^{-1}(p)))$$

Therefore,

$$L(p)^2 = \frac{\lambda_{1,X}}{\lambda_X}L_1(p)^2 + \frac{\lambda_{2,X}}{\lambda_X}L_2(p)^2$$

An analogous argument holds for $p \leq p_0$.

The second statement follows from the fact that that when only one of $L_1(p)$ or $L_2(p)$ is nonzero, we must have that either $L(p) = \sqrt{\frac{\lambda_{1,X}}{\lambda_X}}L_1(p)$ or $L(p) = \sqrt{\frac{\lambda_{2,X}}{\lambda_X}}L_2(p)$. □

C.3.10 Omitted Proof of Proposition 5.5.3

Restatement (Proposition 5.5.3)**.** *Let $p_{min} < p_{max}$ be two arbitrary exchange rates, and let $\psi(p_X, p_Y) = 1$ if and only if $0 \leq p_X \leq P_X$, $0 \leq p_Y \leq P_Y$, and $p_{min} \leq p_X/p_Y \leq p_{max}$, and 0 otherwise. The allocation $L(\cdot)$ that maximizes the fraction of successful trades is the allocation implied by a concentrated liquidity position with price range defined by p_{min} and p_{max}.*

Proof. A concentrated liquidity position trades exactly as a constant product market maker within its price bounds p_{min} and p_{max}, and makes no trades outside of that range.

By Lemma 5.4.10, the optimal $L(p)$ is 0 for p outside of the range $[p_{min}, p_{max}]$. Inside that range, by Proposition 5.4.14, $L(p)$ differs from the optimal liquidity allocation for the constant product market maker by a constant, multiplicative factor (the same factor for every p). Thus, the resulting liquidity allocation has the same behavior as a concentrated liquidity position. □

C.3.11 Omitted Proof of Proposition 5.5.5

Restatement (Proposition 5.5.5)**.** *The belief function $\psi(p_X, p_Y) = \left(\frac{p_X}{p_Y}\right)^{\frac{\alpha-1}{\alpha+1}}$ when $(p_X, p_Y) \in (0, P_X] \times (0, P_Y]$ and 0 otherwise corresponds to the weighted product market maker $f(x, y) = x^\alpha y$.*

Proof. This trading function gives the relation $p = \frac{y\alpha}{x}$ and thus $\mathcal{Y}(p) = p^{\frac{\alpha}{\alpha+1}}(\frac{K}{\alpha^\alpha})^{\frac{1}{\alpha}}$, for $K = f(\hat{X}, \hat{Y})$ and $\hat{X}, \hat{Y}$ is some initial state of the CFMM reserves.

Thus, as defined by the trading function, $L(p) = p^{\frac{\alpha}{\alpha+1}}\frac{\alpha}{\alpha+1}\left(\frac{K}{\alpha^\alpha}\right)^{\frac{1}{\alpha+1}}$.

Corollary 5.4.16 shows that a belief that leads to this liquidity allocation is

$$\frac{L(p)^2}{p} = p^{-1}p^{\frac{2\alpha}{\alpha+1}}\left(\frac{\alpha}{\alpha+1}\right)^2\left(\frac{K}{\alpha^\alpha}\right)^{\frac{2}{\alpha+1}} = p^{\frac{\alpha-1}{\cdot}\alpha+1}C$$

on the rectangle $(0, P_X] \times (0, P_Y]$ and 0 elsewhere, for some constant C. The result follows by rescaling the belief function (Corollary 5.4.6). □

C.3.12 Omitted Proof of Proposition 5.5.6

Restatement (Proposition 5.5.6)**.** *The optimal trading function to minimize the expected CFMM inefficiency for the belief* $\psi(p_X, p_Y) = \frac{p_X p_Y}{(p_X + p_Y)^2}$ *when* $(p_X, p_Y) \in (0, P_X] \times (0, P_Y]$ *and* 0 *otherwise, is* $f(x, y) = 2 - e^{-x} - e^{-y}$.

Proof. This trading function implies the relationship $p = e^{y-x}$.

Combining this with the equation $e^{-y} + e^{-x} = K$ (for some constant K) gives $(1 + p)e^{-y} = K$ and thus $\mathcal{Y}(p) = \ln(\frac{1+p}{K})$. From the definition of liquidity, we obtain $L(p) = \frac{p}{1+p}$.

Corollary 5.4.16 shows that a belief function that leads to this liquidity allocation is (with $p = p_X/p_Y$)

$$\frac{L(p)^2}{p} = \frac{p}{(1+p)^2} = \frac{p_X p_Y}{(p_X + p_Y)^2}$$

for $(p_X, p_Y) \in (0, P_X] \times (0, P_Y]$ and 0 elsewhere. The result follows by rescaling the belief function (Corollary 5.4.6). □

C.3.13 Omitted Proof of Proposition 5.6.4

Restatement (Proposition 5.6.4)**.** *The expected future value of the CFMM's reserves, as per belief* $\psi(p_X, p_Y)$, *is*

$$\nu(\psi) = \frac{1}{N_\psi} \iint\limits_{p_X, p_Y} \psi(p_X, p_Y) \left(p_X \mathcal{X}(p_X/p_Y) + p_Y \mathcal{Y}(p_X/p_Y)\right) dp_X \; dp_Y$$

$$= \frac{1}{N_\psi} \int\limits_0^\infty \left(\frac{L(p)}{p^2} \iint\limits_{p_X, p_Y} p_X \psi(p_X, p_Y) \mathbb{1}_{\{\frac{p_X}{p_Y} \leq p\}} dp_X dp_Y + \frac{L(p)}{p} \iint\limits_{p_X, p_Y} p_Y \psi(p_X, p_Y) \mathbb{1}_{\{\frac{p_X}{p_Y} \geq p\}} dp_X dp_Y \right) dp$$

where $\mathcal{X}(p)$ *and* $\mathcal{Y}(p)$ *denote the amounts of* X *and* Y *held in the reserves at spot exchange rate* p, *and* $\mathbb{1}_E$ *is the characteristic function of the event* E.

This expression for $\nu(\psi)$ *is a linear function of each* $L(p)$.

Proof.

$$\frac{1}{N_\psi}\int\limits_{p_X,p_Y}\psi(p_X,p_Y)\left(p_X\mathcal{X}(p_X/p_Y)\right)+p_Y\mathcal{Y}(p_X/p_Y))\,dp_X\ \ dp_Y$$

$$=\frac{1}{N_\psi}\int\limits_{p_X,p_Y}\psi(p_X,p_Y)\left(p_X\int\limits_{p_X/p_Y}^{\infty}\frac{L(p)}{p^2}dp+p_Y\int\limits_{0}^{p_X/p_Y}\frac{L(p)}{p}dp\right)dp_X\ \ dp_Y$$

$$=\frac{1}{N_\psi}\int\limits_0^\infty\left(\frac{L(p)}{p^2}\iint\limits_{p_X,p_Y}p_X\psi(p_X,p_Y)\mathbb{1}_{\{\frac{p_X}{p_Y}\le p\}}dp_Xdp_Y+\frac{L(p)}{p}\iint\limits_{p_X,p_Y}p_Y\psi(p_X,p_Y)\mathbb{1}_{\{\frac{p_X}{p_Y}\ge p\}}dp_Xdp_Y\right)dp$$

The first equation follows by substitution of the equations in Observation 5.3.10.

Note that for any p_X, p_Y, the term $\frac{L(p)}{p^2}$ for any p appears in the integral $\int_{p_X/p_Y}^{\infty}\frac{L(p)dp}{p^2}$ if and only if $p \ge p_X/p_Y$. The result follows from rearranging the integral to group terms by $L(p)$. □

C.3.14 Omitted Proof of Theorem 5.6.7

Restatement (Theorem 5.6.7)**.** *Let $L_1(p)$ be the optimal liquidity allocation that maximizes fee revenue — the solution to the optimization problem for the objective of minimizing the following:*

$$-\frac{\delta}{N_\psi}\iint_{p_X,p_Y}rate_\delta(p_X,p_Y)\psi(p_X,p_Y)\left(1-\frac{s}{p_YL(p_X/p_Y)}\right)dp_X\ \ dp_Y$$

Let $L_2(p)$ be the optimal liquidity allocation that maximizes fee revenue while accounting for divergence loss — the solution to the optimization problem for the objective of minimizing the following:

$$-\nu(\psi)-\frac{\delta}{N_\psi}\iint_{p_X,p_Y}rate_\delta(p_X,p_Y)\psi(p_X,p_Y)\left(1-\frac{s}{p_YL(p_X/p_Y)}\right)dp_X\ \ dp_Y$$

Let $X_1=\int_{p_0}^{\infty}\frac{L_1(p)}{p^2}dp$ and $X_2=\int_{p_0}^{\infty}\frac{L_2(p)}{p^2}dp$ be the optimal initial quantities of X for the above two problems respectively.

Then there exists some $p_1>p_0$ such that for $p_0\le p\le p_1$, $\frac{L_1(p)}{X_1}\ge\frac{L_2(p)}{X_2}$ and for $p\ge p_1$, $\frac{L_1(p)}{X_1}\le\frac{L_2(p)}{X_2}$. An analogous statement holds for the allocations of Y.

Proof. Define $\varphi'(\theta)=\delta\int_r rate_\delta(r\cos(\theta),r\sin(\theta)\psi(r\cos(\theta),r\sin(\theta))dr$.

The KKT conditions for the first problem give the following (nearly identically to those in Lemma 5.4.11, just using $L_1(\cdot)$ in place of $L(\cdot)$):

1. For all p with $p\ge p_0$, $\frac{\lambda_X}{p^2}=\frac{1}{L_1(p)^2}\varphi'(\cot^{-1}(p))\sin(\cot^{-1}(p))+\lambda_{L_1(p)}$.

2. For all p with $p\le p_0$, $\frac{\lambda_Y}{p}=\frac{1}{L_1(p)^2}\varphi'(\cot^{-1}(p))\sin(\cot^{-1}(p))+\lambda_{L_1(p)}$.

3. $\lambda_X=P_X\lambda_B$ and $\lambda_Y=P_Y\lambda_B$.

Observe that the derivative, with respect to $L(p)$, of the divergence loss, is

$$\kappa(p) = \frac{1}{N_\psi}\left(\frac{1}{p^2}\iint\limits_{p_X,p_Y} p_X\psi(p_X,p_Y)\mathbb{1}_{\{\frac{p_X}{p_Y}\leq p\}}dp_X\ dp_Y + \frac{1}{p}\iint\limits_{p_X,p_Y} p_Y\psi(p_X,p_Y)\mathbb{1}_{\{\frac{p_X}{p_Y}\geq p\}}dp_X\ dp_Y\right).$$

Computing the KKT conditions for the second problem gives the following:

1. For all p with $p \geq p_0$, $\frac{\lambda_X}{p^2} = \frac{1}{L_2(p)^2}\varphi'(\cot^{-1}(p))\sin(\cot^{-1}(p)) + \kappa(p) + \lambda_{L_2(p)}$.
2. For all p with $p \leq p_0$, $\frac{\lambda_Y}{p} = \frac{1}{L_2(p)^2}\varphi'(\cot^{-1}(p))\sin(\cot^{-1}(p)) + \kappa(p) + \lambda_{L_2(p)}$.

As defined in the theorem statement, $X_1 = \int_{p_0}^{\infty}\frac{L_1(p)}{p^2}dp$ and $X_2 = \int_{p_0}^{\infty}\frac{L_2(p)}{p^2}dp$ are the optimal initial quantities of X. Normalizing L_1 and L_2 by X_1 and X_2 respectively gives the equation

$$\int_{p_0}^{\infty}\frac{L_1(p)}{X_1p^2}dp = \int_{p_0}^{\infty}\frac{L_2(p)}{X_2p^2}dp \tag{C.1}$$

This implies that, when normalized by X_1 and X_2, $p \geq p_0$, and $\varphi'(\cot^{-1}(p)) \neq 0$, we have that

$$\begin{aligned}
L_2(p) &= \sqrt{\frac{\varphi'(\cot^{-1}(p))\sin(\cot^{-1}(p))}{\frac{\lambda_X}{p^2} - \kappa(p)}} \\
&= \sqrt{\frac{\varphi'(\cot^{-1}(p))\sin(\cot^{-1}(p))}{\frac{\lambda_X}{p^2}} * \frac{\frac{\lambda_X}{p^2}}{\frac{\lambda_X}{p_2} - \kappa(p)}} \\
&= L_1(p)\frac{X_2}{X_1}\sqrt{\frac{\lambda_X}{\lambda_X - p^2\kappa(p)}}
\end{aligned}$$

A similar argument shows that when $p \leq p_0$,

$$L_2(p) = L_1(p)\frac{Y_2}{Y_1}\sqrt{\frac{\lambda_Y}{\lambda_Y - p\kappa(p)}}$$

Arithmetic calculation gives that

$$\begin{aligned}
\kappa(p)p^2 = \frac{1}{N_\psi}\int_r\int_{\theta'=\theta}^{\pi/2} & r^2\cos(\theta')\psi(r\cos(\theta'), r\sin(\theta'))dr\ \ d\theta' \\
& + \cot(\theta)\int_r\int_{\theta'=0}^{\theta} r^2\sin(\theta')\psi(r\cos(\theta'), r\sin(\theta'))dr\ d\theta'
\end{aligned}$$

and thus that

$$\frac{d(\kappa(\cot(\theta))\cot(\theta)^2)}{d\theta} = -\frac{\csc^2(\theta)}{N_\psi}\int_r\int_{\theta'=0}^{\theta} r^2\sin(\theta')\psi(r\cos(\theta'), r\sin(\theta'))dr\ d\theta' \leq 0$$

$\kappa(\cot(\theta))\cot(\theta)^2$ is therefore decreasing in θ, so $\kappa(p)p^2$ is increasing in p (since $p = \cot(\theta)$). Therefore, $\sqrt{\frac{\lambda_X}{\lambda_X - p^2\kappa(p)}}$ increases as p goes to ∞.

By an analogous argument,

$$\kappa(p)p = \frac{\tan(\theta)}{N_\psi} \int_r \int_{\theta'=\theta}^{\pi/2} r^2 \cos(\theta')\psi(r\cos(\theta'), r\sin(\theta'))dr \; d\theta' \\ + \int_r \int_{\theta'=0}^{\theta} r^2 \sin(\theta')\psi(r\cos(\theta'), r\sin(\theta'))dr \; d\theta'$$

and thus

$$\frac{d(\kappa(\cot(\theta))\cot(\theta))}{d\theta} = \frac{\sec^2(\theta)}{N_\psi} \int_r \int_{\theta'=\theta}^{\pi/2} r^2 \sin(\theta')\psi(r\cos(\theta'), r\sin(\theta'))dr \; d\theta' \geq 0$$

$\kappa(\cot(\theta))\cot(\theta)$ is therefore increasing in θ, so $\kappa(p)p$ increases as p decreases.

Therefore, $\sqrt{\frac{\lambda_Y}{\lambda_Y - p\kappa(p)}}$ increases as p goes to 0.

Equation C.1 implies that the quantities $\frac{L_1(p)}{X_1 p^2}$ and $\frac{L_2(p)}{X_2 p^2}$ integrate to the same value, but $L_2(\cdot)$ increases strictly more quickly than $L_1(\cdot)$, so there must be a point $p_1 > p_0$ beyond which $\frac{L_1(p)}{X_1} \leq \frac{L_2(p)}{X_2}$.

An analogous argument holds for $p < p_0$. $\square$

Appendix D

Chapter 6 Supporting Information

D.1 Existence of Equilibra with Positive Valuations

Assumption D.1.1. *For every asset $\mathcal{A}$ and every set of prices p, if $p_{\mathcal{A}} = 0$, then there exists at least one agent who always has positive marginal utility for $\mathcal{A}$ (no matter how much $\mathcal{A}$ the agent purchases).*

We also make a standard set of assumptions (e.g. as in [78]) on utility functions of agents in an Arrow-Debreu exchange market.

Assumption D.1.2. *A utility function $u(\cdot)$ satisfies the following properties.*

1 $u(\cdot)$ is continuous.

2 For all endowments x, there exists x' with $u(x) < u(x')$.

3 For any x, x' with $u(x) < u(x')$ and any $0 < t < 1$, $u(x) < u(tx + (1-t)x')$.

Lemma D.1.3. *If Assumption D.1.1 holds in some Arrow-Debreu exchange market consisting of CFMMs with supply functions satisfying Assumption 6.5.3 and agents with utility functions satisfies Assumption D.1.2, then there always exists an equilibrium of that market and every equilibrium has a positive valuation on every asset.*

Proof. Let $\hat{e} = (\sum_i e_i) + (1, ..., 1)$ be a vector of assets strictly larger than the total amount of each asset available in the market, and let $E = \{x | 0 \leq x \leq \hat{e}\}$. Let P be the price simplex on $\mathfrak{A}$ (P=$\{p | \sum_{\mathfrak{A}} p_{\mathcal{A}} = 1\}$).

Consider the following game. There is one player for each agent in the exchange market and one "market" player, for a combined state space of $E \times ... \times E \times P$. Clearly, this state space is compact, convex, and nonempty.

Given the set of prices p, each agent player picks a utility-maximizing set of goods x_i subject to resource constraints (specifically, for each agent i, x_i maximizes $u_i(\cdot)$ subject to $p \cdot x \leq p \cdot e_i$ and $x_i \in E$), receiving payoff $u_i(x_i)$. The market player chooses a set of prices $\hat{p}$, for payoff $\sum_i (e_i - x_i) \cdot p$.

Define $A_i(p) = \{x | x \in E, p \cdot x \leq p \cdot e_i\}$ be the set of utility-maximizing sets of goods for agent i. Quasi-concavity of $u_i(\cdot)$ implies that $A_i(p)$ is convex, $A_i(\cdot)$ cannot be nonempty. It suffices to show that $A_i(\cdot)$ is a continuous function for each agent i.

Let $p_1, p_2 ...$ and $x_1, x_2, ...$ be any sequences of prices and demand responses converting to p and x, respectively, with $x_j \in A_i(p_j)$ for all $j \in \mathbb{Z}_+$. Let $r_j = p_j \cdot x_j$ for each j. Naturally, the sequence $r_1, r_2, ...$ must converge to $r = p \cdot x$, and the sequence $u_i(x_1), u_i(x_2) ...$ must converge to $u_i(x)$.

Consider a sequence $\{x'_j\}$ converging to x' with $p_j \cdot x'_j = r_j$ and $x' \in A_i(p)$. It must be the case that $u_i(x_j) \geq u_i(x'_j)$. Because $\{u_i(x'_j)\}$ converges to $u_i(x')$, we must have that $u_i(x) \geq u_i(x')$, so $x \in A_i(p)$ and thus $A_i(\cdot)$ must be continuous.

If some agent is a CFMM defined by a supply function (instead of an agent maximizing a utility function), it suffices that the CFMM's action space, implicitly defined by the supply function, satisfies the same conditions. These must hold for any CFMM Supply function satisfying Assumption 6.5.3.

It follows from Kakutani's fixed point theorem that there must exist a fixed point of this game; that is, there exists a p and an x_i for each i such that $x_i \in A_i(p)$ and $(\sum_i x_i)_{\mathcal{A}} \leq 0$, with the inequality tight if $p_{\mathcal{A}} > 0$ for all assets $\mathcal{A}$.

By Assumption D.1.1, if $p_{\mathcal{A}} = 0$ for some asset $\mathcal{A}$, then there exists an agent i for which every $x_i \in A_i(p)$ has $(x_i)_{\mathcal{A}} = \hat{e}_{\mathcal{A}}$. But then the demand for $\mathcal{A}$ must exceed the available supply, so $(\sum_i x_i)_{\mathcal{A}} > 0$, a contradiction. Analogously, it must be the case that for every asset $\mathcal{A}$ and every agent i, it must be the case that $(x_i)_{\mathcal{A}} < \hat{e}_{\mathcal{A}}$, and thus (by parts 2 and 3 of Assumption D.1.2) x_i maximizes $u_i(\cdot)$ subject to $x_i \cdot p \leq e_i$, $x_i \geq 0$ (i.e. without the restriction that $x_i \in E$).

□

D.2 Trading Rule U and Parallel Running

We show here that in the case where a market is not especially sparse, Trading Rule U is the only trading rule that eliminates parallel running. To be specific, we say that a batch is sufficiently *large* if there is a market sell order trading between every asset pair that can be traded on a CFMM in the batch.

Definition D.2.1 (Large Batch). *A batch instance has a set $\mathcal{C}$ of CFMMs. Let CFMM $c \in \mathcal{C}$ trade in the set of assets $\mathfrak{A}_c$. Denote the set of asset pairs that can be traded on some CFMM in $\mathcal{C}$ by $Pairs(\mathcal{C}) = \{(A, B) | \exists c \in \mathcal{C}, s.t.\ (A, B) \in \mathfrak{A}_c\}$.*

We say that the batch instance is large if there exists a market sell order for every asset pair in $Pairs(\mathcal{C})$.

We first define a property of equilibrium computation algorithm which is very natural but is required for our result in this subsection.

Definition D.2.2 (Split Invariance)**.** *Consider two batch instances.*

1. *There is a set $\mathcal{C}$ of CFMMs and a set $\mathcal{L}$ of limit sell orders.*
2. *There is a set $\mathcal{C}$ of CFMMs and a set $\mathcal{L} \cup \{\tilde{l}_1, \tilde{l}_2\} \setminus \{l\}$ of limit sell orders, where $l = (\mathcal{A}, \mathcal{B}, k, r)$, $\tilde{l}_1 = (\mathcal{A}, \mathcal{B}, t, r)$ and $\tilde{l}_2 = (\mathcal{A}, \mathcal{B}, k - t, r)$, where $\mathcal{A}, \mathcal{B}$ are any two assets, $k > t > 0$, and $r \geq 0$.*

An equilibrium computation algorithm is split-invariant if it always computes the same asset valuations in the above two cases.

Split invariance is a natural condition for algorithms whose output depend on the aggregate behavior of the limit orders, and never on the exact sizes of individual limit orders. Many natural algorithms (such as [127, 90]) satisfy this condition.

We have the following result.

Theorem D.2.3. *In a batch exchange using a split-invariant equilibrium computation algorithm, JPD is necessary to ensure that parallel running is impossible on large batch instances.*

Proof. Suppose that JPD is not guaranteed by a batch exchange, and that there is some large batch input $\mathcal{X}$ where JPD is not satisfied.

As such, there must exist some CFMM c in $\mathcal{X}$ trading in some assets $\mathcal{A}$ and $\mathcal{B}$ for which the post-batch CFMM spot exchange rate $\hat{q}_{\mathcal{A},\mathcal{B}}$ differs from the equilibrium batch exchange rate $\hat{p}_{\mathcal{A},\mathcal{B}} = \frac{p_{\mathcal{A}}}{p_{\mathcal{B}}}$. Without loss of generality, suppose that $\hat{q}_{\mathcal{A},\mathcal{B}} < \hat{p}_{\mathcal{A},\mathcal{B}}$, and denote $\delta_{\mathcal{A},\mathcal{B}} = \hat{p}_{\mathcal{A},\mathcal{B}} - \hat{q}_{\mathcal{A},\mathcal{B}} > 0$. Let l be the market order that sells A for B in the batch, and denote the amount of $\mathcal{A}$ sold by this order by $s_{\mathcal{A},\mathcal{B}}$.

Define $\varepsilon_{\mathcal{A},\mathcal{B}}$ to be the amount of $\mathcal{A}$ sold by c as its spot price moves from $\hat{q}_{\mathcal{A},\mathcal{B}}$ to $\hat{p}_{\mathcal{A},\mathcal{B}}$ when operating as a standalone CFMM. It must be the case that $\varepsilon_{\mathcal{A},\mathcal{B}} > 0$ (or else the above scenario would not violate JPD).

Now, consider an alternate batch instance $\mathcal{X}'$, which differs from $\mathcal{X}$ only in that the order l is replaced with two market orders $\tilde{l}_1$ and $\tilde{l}_2$, of sizes $\varepsilon_{\mathcal{A},\mathcal{B}}$ and $s_{\mathcal{A},\mathcal{B}} - \varepsilon_{\mathcal{A},\mathcal{B}}$ respectively. Because the batch uses a split-invariant equilibrium computation algorithm, the exchange rate in $\mathcal{X}'$ between $\mathcal{A}$ and $\mathcal{B}$ is the same as that computed in $\mathcal{X}'$, namely, $\hat{p}_{\mathcal{A},\mathcal{B}}$.

Now consider an additional, alternative batch instance $\mathcal{X}''$, which is equal to $\mathcal{X}'$ except that it does not include order $\tilde{l}_1$.

We claim that $\mathcal{X}''$ admits a parallel-running opportunity. A parallel runner (who has complete information of $\mathcal{X}''$ and the equilibrium computation algorithm) can insert a market sell order selling $\varepsilon_{\mathcal{A},\mathcal{B}}$ units of $\mathcal{A}$ for $\mathcal{B}$, thereby creating batch $\mathcal{X}'$. After the batch, CFMM c's spot price will be $\hat{q}_{\mathcal{A},\mathcal{B}} < \hat{p}_{\mathcal{A},\mathcal{B}}$, and the parallel runner can sell some amount of $\mathcal{B}$ less than $\varepsilon_{\mathcal{A},\mathcal{B}}\hat{p}_{\mathcal{A},\mathcal{B}}$ to buy back $\varepsilon_{\mathcal{A},\mathcal{B}}$ units of $\mathcal{A}$ from the CFMM. □

Note that Definition D.2.1 is sufficient but not necessary for the proof of Theorem D.2.3. Informally, many natural market scenarios admit similar parallel-running scenarios when not using JPD.

Bibliography

[1] Balancer swap interface. https://app.balancer.fi/#/trade. Accessed 07/04/2022.

[2] Binance chain docs - fees. https://web.archive.org/web/20200617014623/https://docs.binance.org/guides/concepts/fees.html. Accessed 10/18/2022.

[3] Binance chain docs - introduction. https://web.archive.org/web/20200616190856/https://docs.binance.org/guides/intro.html. Accessed 10/18/2022.

[4] Binance chain docs - match steps and examples. https://web.archive.org/web/20200617065916/https://docs.binance.org/match-examples.html. Accessed 10/18/2022.

[5] Coinbase pricing and fees disclosures. https://web.archive.org/web/20210301043800/https://help.coinbase.com/en/coinbase/trading-and-funding/pricing-and-fees/fees.

[6] Cow protocol overview: The batch auction optimization problem. https://web.archive.org/web/20220614183101/https://docs.cow.fi/off-chain-services/in-depth-solver-specification/the-batch-auction-optimization-problem. Accessed 10/19/2022.

[7] Cowswap faq. https://cowswap.exchange/#/faq#how-does-cowswap-determine-prices. Accessed 2/9/2022.

[8] enaira. https://web.archive.org/web/20230624204115/https://www.enaira.gov.ng/. Accessed 1/13/2022.

[9] Eth2 shard chains. https://web.archive.org/web/20210310043308/https://ethereum.org/en/eth2/shard-chains/.

[10] Ethereum development documentation: Optimistic rollups. https://web.archive.org/web/20230330021451/https://ethereum.org/en/developers/docs/scaling/optimistic-rollups/. Accessed 07/31/2023.

[11] Ethereum development documentation: Zk rollups. https://web.archive.org/web/20230623142748/https://ethereum.org/en/developers/docs/scaling/zk-rollups/. Accessed 07/31/2023.

[12] An exchange protocol for the decentralized web. https://docs.gnosis.io/protocol/docs/introduction1/ and https://github.com/gnosis/dex-research/blob/08204510e3047c533ba9ee42bf24f980d087fa78/dFusion/dfusion.v1.pdf.

[13] Intel oneapi threading building blocks. "https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html". Accessed 5/6/2021.

[14] The maker protocol: Makerdao's multi-collateral dai (mcd) system. https://web.archive.org/web/20220301170838/https://makerdao.com/en/whitepaper/. Accessed 08/02/2023.

[15] Near. https://near.org/.

[16] Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum/tree/release/1.10. Accessed 10/13/2022.

[17] The penumbra protocol: Sealed-bid batch swaps. https://web.archive.org/web/20220614034906/https://protocol.penumbra.zone/main/zswap/swap.html. Accessed 10/19/2022.

[18] Polygon lightpaper: Ethereum's internet of blockchains. https://polygon.technology/lightpaper-polygon.pdf. Accessed 12/6/2021.

[19] Serum: Faster, cheaper, and more powerful defi. https://web.archive.org/web/20220516140755/https://www.projectserum.com/. Accessed 12/6/2021.

[20] Solana documentation: Rent. https://web.archive.org/web/20220903121658/https://docs.solana.com/implemented-proposals/rent.

[21] Starkex. https://starkware.co/product/starkex/.

[22] Starkex applications. https://web.archive.org/web/20230521060655/https://docs.starkware.co/starkex/architecture/starkware-exchange-applications.html.

[23] Stellar. https://www.stellar.org/.

[24] A technical introduction to the serum dex. https://web.archive.org/web/20221112173940/https://docs.google.com/document/d/1isGJES4jzQutI0GtQGuqtrBUqeHxl_xJNXdtOv4SdII/edit. Accessed 12/11/22.

[25] Uniswap swap interface. https://app.uniswap.org/#/swap?chain=mainnet. Accessed 07/04/2022.

[26] USDC: the world's leading digital dollar stablecoin. https://web.archive.org/web/20230726103908/https://www.circle.com/en/usdc. Accessed 08/02/2023.

[27] wasm3: The fastest webassembly interpreter, and the most universal runtime. https://github.com/wasm3/wasm3.

[28] Hyperconomy. https://web.archive.org/web/20230124140406/https://github.com/patrick-layden/HyperConomy and https://dev.bukkit.org/projects/hyperconomy and https://web.archive.org/web/20220427213437/https://dev.bukkit.org/projects/hyperconomy/pages/information, 2012. Accessed 08/03/2023.

[29] The battery powered picklejar heads. https://web.archive.org/web/20230201115717/https://blog.picklejarheads.com/about/, 2013.

[30] Loopring: A decentralized token exchange protocol. September 2018.

[31] Aave protocol whitepaper, version 1.0. https://web.archive.org/web/20230426224123/https://github.com/aave/aave-protocol/blob/master/docs/Aave_Protocol_Whitepaper_v1_0.pdf, January 2020.

[32] Uniswap v2 smart contract source. https://web.archive.org/web/20220324172913/https://github.com/Uniswap/v2-core/blob/1136544ac842ff48ae0b1b939701436598d74075/contracts/UniswapV2Pair.sol, 2020.

[33] Uniswapv2router02.sol. https://web.archive.org/web/20220605141506/https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol, Jun 2020.

[34] Gpv2 objective criterion. https://web.archive.org/web/20211019155516/https://forum.gnosis.io/t/gpv2-objective-criterion/1254, April 2021. Accessed 04/30/2021.

[35] Loopring 3 design doc. https://web.archive.org/web/20220411224154/https://github.com/Loopring/protocols/blob/master/packages/loopring_v3/DESIGN.md#results, 2021.

[36] The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. https://web.archive.org/web/20221020032330/https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf, August 2022. Accessed 10/20/22.

[37] Compound iii docs. https://web.archive.org/web/20230715092703/https://docs.compound.finance/, 2022.

[38] Cosmwasm documentation: Actor model for contract calls. https://web.archive.org/web/20220811195538/https://docs.cosmwasm.com/docs/1.0/architecture/actor/, 2022.

[39] Loopring protocol. https://web.archive.org/web/20220409050852/https://loopring.org/#/protocol, April 2022.

[40] Near documentation: Cross-contract calls. https://web.archive.org/web/20221021051657/https://docs.near.org/develop/contracts/crosscontract, 2022.

[41] Solana documentation: Accounts. https://web.archive.org/web/20221110215034/https://docs.solana.com/developing/programming-model/accounts/, 2022.

[42] United States Securities and Exchange Commission v. Lawrence Billimek and Alan Williams, case 1:22-cv-10542, December 2022. https://www.sec.gov/files/litigation/complaints/2022/comp-pr2022-228.pdf.

[43] Wasm-instrument: Instrument and transform wasm modules. https://github.com/paritytech/wasm-instrument, 2022.

[44] Compound finance. https://web.archive.org/web/20230414212550/https://compound.finance/, 2023. Accessed 04/14/2023.

[45] DefiLlama: Dexs volume. https://web.archive.org/web/20230628135700/https://defillama.com/dexs, 2023.

[46] Dydx v4 technical overview. https://web.archive.org/web/20230516231337/https://dydx.exchange/blog/v4-technical-architecture-overview, 2023.

[47] Securities and Exchange Commission v. Binance Holdings Limited, BAM Trading Services Inc., BAM Management US Holdings Inc., and Changpeng Zhao, civil action no. 1:23-cv-01599, June 2023. https://www.sec.gov/files/litigation/complaints/2023/comp-pr2023-101.pdf.

[48] Securities and Exchange Commission v. Celsius Network Limited and Alexander "Alex" Mashinsky, no. 1:23-cv-6003. https://www.sec.gov/files/litigation/complaints/2023/comp-pr2023-133.pdf, July 2023.

[49] Securities and Exchange Commission v. Coinbase Inc. and Coinbase Global, Inc., case 1:23-cv-04738, June 2023. https://www.sec.gov/files/litigation/complaints/2023/comp-pr2023-102.pdf.

[50] Securities and Exchange Commission v. Terraform Labs Pte Ltd. and Do Hyeong Kwon, no. 1:23-cv-1346, February 2023. https://www.sec.gov/files/litigation/complaints/2023/comp-pr2023-32.pdf.

[51] Synthetix litepaper, version 1.6. https://web.archive.org/web/20230327133255/https://docs.synthetix.io/synthetix-protocol/the-synthetix-protocol/synthetix-litepaper, January 2023.

[52] United States of America v. Alexander Mashinsky and Roni Cohen-Pavon, no. 23 crim 347, July 2023. https://www.justice.gov/d9/2023-07/u.s._v._mashinsky_and_cohen-pavon_indictment.pdf.

[53] United States of America v. Samuel Bankman-Fried, a/k/a "SBF", case 1:22-cr-00673-lak, February 2023. https://storage.courtlistener.com/recap/gov.uscourts.nysd.590940/gov.uscourts.nysd.590940.80.0.pdf.

[54] Michael Ackerman, Sul-Young Choi, Peter Coughlin, Eric Gottlieb, and Japheth Wood. Elections with partially ordered preferences. *Public Choice*, 157:145–168, 2013.

[55] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020.

[56] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. Technical report, Tech. rep., Uniswap, 2021.

[57] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616, 2020.

[58] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, et al. Status report on the third round of the nist post-quantum cryptography standardization process. *US Department of Commerce, NIST*, 2022.

[59] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The opbench ethereum opcode benchmark framework: Design, implementation, validation and experiments. *Performance Evaluation*, 146:102168, 2021.

[60] Aurélien Alfonsi, Antje Fruth, and Alexander Schied. Optimal execution strategies in limit order books with general shape functions. *Quantitative finance*, 10(2):143–157, 2010.

[61] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, pages 1–45, 2020.

[62] Yakov Amihud and Haim Mendelson. Dealership market: Market-making with inventory. *Journal of financial economics*, 8(1):31–53, 1980.

[63] Yakov Amihud and Haim Mendelson. Asset pricing and the bid-ask spread. *Journal of financial Economics*, 17(2):223–249, 1986.

[64] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[65] Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 80–91, 2020.

[66] Guillermo Angeris, Alex Evans, and Tarun Chitra. Replicating monotonic payoffs without oracles. *arXiv preprint arXiv:2111.13740*, 2021.

[67] Guillermo Angeris, Alex Evans, and Tarun Chitra. Replicating market makers. *Digital Finance*, pages 1–21, 2023.

[68] Guillermo Angeris, Alex Evans, Tarun Chitra, and Stephen Boyd. Optimal routing for constant function market makers. In *Proceedings of the 23rd ACM Conference on Economics and Computation*, pages 115–128, 2022.

[69] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *Cryptoeconomic Systems Journal*, 2021.

[70] Parwat Singh Anjana, Hagit Attiya, Sweta Kumari, Sathya Peri, and Archit Somani. Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory. In *International Conference on Networked Systems*, pages 77–93. Springer, 2020.

[71] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 83–92. IEEE, 2019.

[72] Jun Aoyagi. Liquidity provision by automated market makers. *Available at SSRN 3674178*, 2020.

[73] Matteo Aquilina, Eric Budish, and Peter O'neill. Quantifying the high-frequency trading "arms race". *The Quarterly Journal of Economics*, 137(1):493–564, 2022.

[74] Matteo Aquilina, Eric B Budish, and Peter O'Neill. Quantifying the high-frequency trading" arms race": A simple new methodology and estimates. Technical report, Working Paper, 2020.

[75] Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[76] Kenneth J Arrow. A difficulty in the concept of social welfare. *Journal of political economy*, 58(4):328–346, 1950.

[77] Kenneth J Arrow, Henry D Block, and Leonid Hurwicz. On the stability of the competitive equilibrium, ii. *Econometrica: Journal of the Econometric Society*, pages 82–109, 1959.

[78] Kenneth J Arrow and Gerard Debreu. Existence of an equilibrium for a competitive economy. *Econometrica: Journal of the Econometric Society*, pages 265–290, 1954.

[79] Jean-Philippe Aumasson and Markku-Juhani O Saarinen. The blake2 cryptographic hash and message authentication code (mac). *RFC 7693*, 2015.

[80] Vivek Bagaria, Joachim Neu, and David Tse. Boomerang: Redundancy improves latency and throughput in payment-channel networks. In *International Conference on Financial Cryptography and Data Security*, pages 304–324. Springer, 2020.

[81] Balancer Labs. Balancer v2 documentation: Weighted pools. https://web.archive.org/web/20220625080037/https://docs.balancer.fi/products/balancer-pools/weighted-pools, 2021.

[82] Markus Baldauf and Joshua Mollner. High-frequency trading and market performance. *The Journal of Finance*, 75(3):1495–1526, 2020.

[83] European Central Bank. The case for a digital euro: key objectives and design considerations. https://www.ecb.europa.eu/pub/pdf/other/key_objectives_digital_euro~f11592d6fb.en.pdf, July 2022.

[84] European Central Bank. Digital euro - prototype summary and lessons learned. https://www.ecb.europa.eu/pub/pdf/other/ecb.prototype_summary20230526~71d0b26d55.en.pdf, May 2023.

[85] European Central Bank. Successful launch of new t2 wholesale payment system. https://web.archive.org/web/20230323155412/https://www.ecb.europa.eu/press/pr/date/2023/html/ecb.pr230321~f5c7bddf6d.en.html, March 2023.

[86] Yogev Bar-On and Yishay Mansour. Uniswap liquidity provision: An online learning approach. *arXiv preprint arXiv:2302.00610*, 2023.

[87] Nicolas Barry. Core advancement protocol 46-07: Fee model in smart contracts. https://web.archive.org/web/20221212214353/https://github.com/stellar/stellar-protocol/blob/master/core/cap-0046-07.md, Jun 2022.

[88] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A theory of automated market makers in defi. In *International Conference on Coordination Languages and Models*, pages 168–187. Springer, 2021.

[89] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A true concurrent model of smart contracts executions. In *International Conference on Coordination Languages and Models*, pages 243–260. Springer, 2020.

[90] Xiaohui Bei, Jugal Garg, and Martin Hoefer. Ascending-price algorithms for unknown markets. *ACM Transactions on Algorithms (TALG)*, 15(3):1–33, 2019.

[91] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. 2018.

[92] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. Deep-fri: sampling outside the box improves soundness. *arXiv preprint arXiv:1903.12243*, 2019.

[93] Walter Benjamin. The work of art in the age of mechanical reproduction, 1935.

[94] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[95] Philippe Bergault, Louis Bertucci, David Bouba, and Olivier Guéant. Automated market makers: Mean-variance analysis of lps payoffs and design of pricing functions. *arXiv preprint arXiv:2212.00336*, 2022.

[96] Alex Biryukov, Gleb Naumenko, and Sergei Tikhomirov. Analysis and probing of parallel channels in the lightning network. In *International Conference on Financial Cryptography and Data Security*, pages 337–357. Springer, 2022.

[97] Ivan Bogatyy. Implementing ethereum trading front-runs on the bancor exchange in python. https://web.archive.org/web/20220119154606/https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798, Aug 2017.

[98] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 947–964. IEEE, 2020.

[99] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[100] Lorenz Breidenbach, Christian Cachin, Benedict Chan, Alex Coventry, Steve Ellis, Ari Juels, Farinaz Koushanfar, Andrew Miller, Brendan Magauran, Daniel Moroz, et al. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. https://web.archive.org/web/20230504134314/https://research.chain.link/whitepaper-v2.pdf, 2021. Accessed 08/03/2023.

[101] Eric Budish. Response to esma's call for evidence: "periodic auctions for equity instruments" (esma70-156-785). https://ericbudish.org/wp-content/uploads/2022/03/response_esmas_call_evidence_periodic_auctions.pdf, January 2019. Accessed 10/17/2022.

[102] Eric Budish, Peter Cramton, and John Shim. Implementation details for frequent batch auctions: Slowing down markets to the blink of an eye. *American Economic Review*, 104(5):418–24, 2014.

[103] Eric Budish, Peter Cramton, and John Shim. The high-frequency trading arms race: Frequent batch auctions as a market design response. *The Quarterly Journal of Economics*, 130(4):1547–1621, 2015.

[104] Eric B Budish, Peter Cramton, Albert S Kyle, and Jeongmin Lee. Flow trading. *University of Chicago, Becker Friedman Institute for Economics Working Paper*, (2022-82), 2022.

[105] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*, pages 315–334. IEEE, 2018.

[106] Benedikt Bünz and Binyi Chen. Protostar: Generic efficient accumulation/folding for special sound protocols. *Cryptology ePrint Archive*, 2023.

[107] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*, pages 65–97. Springer, 2021.

[108] Vitalik Buterin. A quick explanation of what the point of the eip 2929 gas cost increases in berlin is. https://web.archive.org/web/20211017034159/https://www.reddit.com/r/ethereum/comments/mrl5wg/a_quick_explanation_of_what_the_point_of_the_eip/, April 2021.

[109] Vitalik Buterin. A state expiry and statelessness roadmap. https://web.archive.org/web/20220916204724/https://notes.ethereum.org/@vbuterin/verkle_and_state_expiry_proposal, 2021.

[110] Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 316–333. Springer, 2022.

[111] Agostino Capponi and Ruizhe Jia. The adoption of blockchain-based decentralized exchanges. *arXiv preprint arXiv:2103.08842*, 2021.

[112] Álvaro Cartea, Fayçal Drissi, and Marcello Monga. Decentralised finance and automated market making: Predictable loss and optimal liquidity provision. *Available at SSRN 4273989*, 2022.

[113] Álvaro Cartea and Sebastian Jaimungal. A closed-form execution strategy to target volume weighted average price. *SIAM Journal on Financial Mathematics*, 7(1):760–785, 2016.

[114] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 99)*, pages 173–186, 1999.

[115] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[116] New York Innovation Center. Project cedar: Phase 1 report. https://www.newyorkfed.org/medialibrary/media/nyic/project-cedar-phase-one-report.pdf, 2022.

[117] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *International conference on information security practice and experience*, pages 3–24. Springer, 2017.

[118] Xi Chen, Decheng Dai, Ye Du, and Shang-Hua Teng. Settling the complexity of arrow-debreu equilibria in markets with additively separable utilities. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 273–282. IEEE, 2009.

[119] Xi Chen, Dimitris Paparas, and Mihalis Yannakakis. The complexity of non-monotone markets. *Journal of the ACM (JACM)*, 64(3):1–56, 2017.

[120] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. Forerunner: Constraint-based speculative transaction execution for ethereum (full version). 2021.

[121] Yiling Chen and David M Pennock. A utility framework for bounded-loss market makers. *arXiv preprint arXiv:1206.5252*, 2012.

[122] Yiling Chen and Jennifer Wortman Vaughan. A new understanding of prediction markets via no-regret learning. In *Proceedings of the 11th ACM conference on Electronic commerce*, pages 189–198, 2010.

[123] Graciela Chichilnisky and Geoffrey Heal. Social choice with infinite populations: construction of a rule and impossibility results. *Social Choice and Welfare*, 14:303–318, 1997.

[124] Howard Chu and Symas Corporation. Lightning memory-mapped database manager (lmdb). http://www.lmdb.tech/doc/. Accessed 04/29/2021.

[125] Austin T Clements, M Frans Kaashoek, Nickolai Zeldovich, Robert T Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):1–47, 2015.

[126] Dan Cline, Thaddeus Dryja, and Neha Narula. Clockwork: An exchange protocol for proofs of non front-running.

[127] Bruno Codenotti, Benton McCune, and Kasturi Varadarajan. Market equilibrium via the excess demand function. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 74–83, 2005.

[128] Bruno Codenotti, Sriram V Pemmaraju, and Kasturi R Varadarajan. On the polynomial time computation of equilibria for certain exchange economies. In *SODA*, volume 5, pages 72–81, 2005.

[129] Kalman J Cohen and Robert A Schwartz. An electronic call market: Its design and desirability. *The Electronic Call Auction: Market Mechanism and Trading: Building A Better Stock Market*, pages 55–85, 2001.

[130] CoinGecko. Cryptocurrency prices by market cap. https://web.archive.org/web/20221210015628/https://www.coingecko.com/.

[131] Richard Cole and Lisa Fleischer. Fast-converging tatonnement algorithms for one-time and ongoing market problems. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 315–324, 2008.

[132] Josep M Colomer. Ramon llull: from 'ars electionis'to social choice theory. *Social Choice and Welfare*, 40(2):317–328, 2013.

[133] Marquis de Condorcet and Marquis de Caritat. An essay on the application of analysis to the probability of decisions rendered by a plurality of votes. *Classics of social choice*, pages 91–112, 1785.

[134] Vincent Conitzer and Tuomas Sandholm. Communication complexity of common voting rules. In *Proceedings of the 6th ACM conference on Electronic commerce*, pages 78–87, 2005.

[135] OpenZeppelin Contracts. Erc20.sol. `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol`.

[136] Arthur H Copeland. A reasonable social welfare function. Technical report, mimeo, 1951. University of Michigan, 1951.

[137] Bernard Cornet. Linear exchange economies. *Cahier Eco-Math, Université de Paris*, 1, 1989.

[138] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[139] Simon Cousaert, Jiahua Xu, and Toshiko Matsui. Sok: Yield aggregators in defi. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–14. IEEE, 2022.

[140] Karl-Dieter Crisman, Abraham Holleran, Micah Martin, and Josephine Noonan. Voting on cyclic orders, group theory, and ballots. *arXiv preprint arXiv:2211.04545*, 2022.

[141] John Cullinan, Samuel K Hsiao, and David Polett. A borda count for partially ordered ballots. *Social Choice and Welfare*, 42:913–926, 2014.

[142] B Curtis Eaves. Finite solution of pure trade markets with cobb-douglas utilities. *Economic Equilibrium: Model Formulation and Solution*, pages 226–239, 1985.

[143] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

[144] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[145] Sourav Das, Vinith Krishnan, and Ling Ren. Efficient cross-shard transaction execution in sharded blockchains. *arXiv preprint arXiv:2007.14521*, 2020.

[146] Nikhil R Devanur, Jugal Garg, and László A Végh. A rational convex program for linear arrow-debreu markets. *ACM Transactions on Economics and Computation (TEAC)*, 5(1):1–13, 2016.

[147] Nikhil R Devanur and Vijay V Vazirani. An improved approximation scheme for computing Arrow-Debreu prices for the linear case. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 149–155. Springer, 2003.

[148] Zijing Di, Lucas Xia, Wilson Nguyen, and Nirvan Tyagi. Muxproofs: Succinct arguments for machine computation from tuple lookups. *Cryptology ePrint Archive*, 2023.

[149] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[150] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Computing*, pages 1–17, 2019.

[151] OpenZeppelin Documentation. Erc20. https://docs.openzeppelin.com/contracts/2.x/api/token/erc20. Accessed 12/10/22.

[152] Alexander Domahidi, Eric Chu, and Stephen Boyd. Ecos: An socp solver for embedded systems. In *2013 European Control Conference (ECC)*, pages 3071–3076. IEEE, 2013.

[153] Eric Dorre. Hold on... this doesn't make sense - frequent batch auctions, part i. https://web.archive.org/web/20230809001613/https://highburyassociates.com/blog/wait-a-0100-second-this-doesnt-make-sense-frequent-batch-auctions-part-i, 2020. Accessed 6/3/2023.

[154] Ran Duan and Kurt Mehlhorn. A combinatorial polynomial algorithm for the linear Arrow–Debreu market. *Information and Computation*, 243:112–132, 2015.

[155] Nicholas Economides and Robert A Schwartz. *Electronic call market trading*. Springer, 2001.

[156] Boris A Efimov and Gleb A Koshevoy. A topological approach to social choice with infinite populations. *Mathematical Social Sciences*, 27(2):145–157, 1994.

[157] Michael Egorov. Stableswap-efficient mechanism for stablecoin liquidity. *Retrieved Feb*, 24:2021, 2019.

[158] Edmund Eisenberg and David Gale. Consensus of subjective probabilities: The pari-mutuel method. *The Annals of Mathematical Statistics*, 30(1):165–168, 1959.

[159] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. Eip-721: Erc-721 non-fungible token standard. *Ethereum Improvement Proposals*, 721, 2018.

[160] William Entriken, Dieter Shirley, and Nastassia Sachs. Eip-721: Non-fungible token standard. *Ethereum Improvement Protocols*, 721, 2018.

[161] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: front-running attacks on blockchain. In *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pages 170–189. Springer, 2020.

[162] Alex Evans, Guillermo Angeris, and Tarun Chitra. Optimal fees for geometric mean market makers. In *International Conference on Financial Cryptography and Data Security*, pages 65–79. Springer, 2021.

[163] Brandon Fain, Ashish Goel, Kamesh Munagala, and Nina Prabhu. Random dictators with a random referee: Constant sample complexity mechanisms for social choice. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1893–1900, 2019.

[164] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324*, 2014.

[165] Zhou Fan, Francisco Marmolejo-Cossío, Ben Altschuler, He Sun, Xintong Wang, and David C Parkes. Differential liquidity provision in uniswap v3 and implications for contract design. *arXiv preprint arXiv:2204.00464*, 2022.

[166] Zhou Fan, Francisco Marmolejo-Cossio, Daniel Moroz, Michael Neuder, Rithvik Rao, and David C Parkes. Strategic liquidity provision in uniswap v3. *arXiv preprint arXiv:2106.12033*, 2023.

[167] Matheus VX Ferreira and David C Parkes. Credible decentralized exchange design via verifiable sequencing rules. *arXiv preprint arXiv:2209.15569*, 2022.

[168] Peter C Fishburn. Arrow's impossibility theorem: concise proof and infinite voters. *Journal of Economic Theory*, 2(1):103–106, 1970.

[169] Stellar Development Foundation. Stellar for cbdcs. https://resources.stellar.org/hubfs/Stellar_CBDC_Whitepaper.pdf. Accessed 2/24/2023.

[170] Stellar Development Foundation. Ukraine electronic hryvnia pilot launched by tascombank and bitt on stellar. https://web.archive.org/web/20230401101149/https://www.stellar.org/press-releases/ukraine-electronic-hryvnia-pilot-launched-by-tascombank-and-bitt-on-stellar, December 2021.

[171] Rafael Frongillo, Maneesha Papireddygari, and Bo Waggoner. An axiomatic characterization of cfmms and equivalence to prediction markets. *arXiv preprint arXiv:2302.00196*, 2023.

[172] Drew Fudenberg and Jean Tirole. *Game theory*. MIT press, 1991.

[173] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[174] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. *arXiv preprint arXiv:2201.03749*, 2022.

[175] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310. Springer, 2015.

[176] Jugal Garg, Edin Husić, and László A Végh. Auction algorithms for market equilibrium with weak gross substitute demands and their applications. *arXiv preprint arXiv:1908.07948*, 2019.

[177] Jugal Garg and László A Végh. A strongly polynomial algorithm for linear exchange markets. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 54–65, 2019.

[178] Corey Garriott and Ryan Riordan. Trading on long-term information. In *Proceedings of Paris December 2020 Finance Meeting EUROFIDAI-ESSEC*, 2020.

[179] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

[180] Austin Gerig and David Michayluk. Automated liquidity provision and the demise of traditional market making. *arXiv preprint arXiv:1007.2352*, 2010.

[181] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[182] Lawrence R Glosten and Paul R Milgrom. Bid, ask and transaction prices in a specialist market with heterogeneously informed traders. *Journal of financial economics*, 14(1):71–100, 1985.

[183] Ashish Goel and Geoffrey Ramseyer. Continuous credit networks and layer 2 blockchains: Monotonicity and sampling. In *Proceedings of the 21st ACM Conference on Economics and Computation*, pages 613–635, 2020.

[184] Christopher Goes, Awa Sun Yin, and Adrian Brink. Anoma: Undefining money. 2021.

[185] Mohak Goyal, Geoffrey Ramseyer, Ashish Goel, and David Mazieres. Finding the right curve: Optimal design of constant function market makers. EC '23, page 783–812, New York, NY, USA, 2023. Association for Computing Machinery. DOI 10.1145/3580507.3597688.

[186] F Grafe and J Grafe. On arrow-type impossibility theorems with infinite individuals and infinite alternatives. *Economics Letters*, 11(1-2):75–79, 1983.

[187] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, pages 305–326. Springer, 2016.

[188] Günter Hägele and Friedrich Pukelsheim. Lulls' writings on electoral sytems. 2001.

[189] Runchao Han, Haoyu Lin, and Jiangshan Yu. On the optionality and fairness of atomic swaps. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 62–75, 2019.

[190] Robin Hanson. Combinatorial information market design. *Information Systems Frontiers*, 5(1):107–119, 2003.

[191] Robin Hanson. Logarithmic markets scoring rules for modular combinatorial information aggregation. *The Journal of Prediction Markets*, 1(1):3–15, 2007.

[192] Theodore E Harris. The existence of stationary measures for certain markov processes. In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability*, volume 2, pages 113–124, 1956.

[193] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. Risks and returns of uniswap v3 liquidity providers. *arXiv preprint arXiv:2205.08904*, 2022.

[194] Eyal Hertzog, Guy Benartzi, and Galia Benartzi. Bancor protocol. 2018.

[195] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute Menlo Park, CA, 1973.

[196] Graydon Hoare. Core advancement protocol 53: Smart contract data, Mar 2022. https://github.com/stellar/stellar-protocol/blob/master/core/cap-0053.md.

[197] BIS Innovation Hub. Project helvetia phase ii: Settling tokenised assets in wholesale cbdc. https://www.bis.org/publ/othp45.pdf, 2022. Accessed 2/24/2023.

[198] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[199] Kamal Jain. A polynomial time algorithm for computing an Arrow–Debreu market equilibrium for linear utilities. *SIAM Journal on Computing*, 37(1):303–318, 2007.

[200] Kamal Jain, Mohammad Mahdian, and Amin Saberi. Approximating market equilibria. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 98–108. Springer, 2003.

[201] Jonathan Jove, Geoffrey Ramseyer, and Jay Geng. Core advancement protocol 45: Speedex - pricing. `https://web.archive.org/web/20221013012607/https://github.com/stellar/stellar-protocol/blob/master/core/cap-0045.md`, Feb 2022.

[202] Mudabbir Kaleem, Keshav Kasichainula, Rabimba Karanjai, Lei Xu, Zhimin Gao, Lin Chen, and Weidong Shi. An event driven framework for smart contract execution. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, pages 78–89, 2021.

[203] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1353–1370, 2018.

[204] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 237–250, 2012.

[205] Alireza Kavousi, Duc V Le, Philipp Jovanovic, and George Danezis. Blindperm: Efficient mev mitigation with an encrypted mempool and permutation. *Cryptology ePrint Archive*, 2023.

[206] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

[207] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In *Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop*, pages 3–14, 2022.

[208] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *Cryptology ePrint Archive*, 2021.

[209] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*, pages 451–480. Springer, 2020.

[210] Zoltán Király and Péter Kovács. Efficient implementations of minimum-cost flow algorithms. *arXiv preprint arXiv:1207.6381*, 2012.

[211] Alan P Kirman and Dieter Sondermann. Arrow's theorem, many agents, and invisible dictators. *Journal of Economic Theory*, 5(2):267–277, 1972.

[212] Abhiram Kothapalli and Srinath Setty. Supernova: Proving universal machine executions without universal circuits. *Cryptology ePrint Archive*, 2022.

[213] Abhiram Kothapalli and Srinath Setty. Hypernova: Recursive arguments for customizable constraint systems. *Cryptology ePrint Archive*, 2023.

[214] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*, pages 359–388. Springer, 2022.

[215] Kiyoshi Kuga. Weak gross substitutability and the existence of competitive equilibrium. *Econometrica: Journal of the Econometric Society*, pages 593–599, 1965.

[216] Albert S Kyle. Continuous auctions and insider trading. *Econometrica: Journal of the Econometric Society*, pages 1315–1335, 1985.

[217] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[218] Pi Lanningham and SundaeSwap Team. Sundaeswap fundamentals. https://web.archive.org/web/20220413054236/https://sundaeswap.finance/papers/SundaeSwap-2021-06-01-Fundamentals.pdf, June 2021.

[219] Yi-Tsung Lee, Roberto Riccò, and Kai Wang. Frequent batch auctions vs. continuous trading: Evidence from taiwan. *Continuous Trading: Evidence from Taiwan (November 19, 2020)*, 2020.

[220] Alfred Lehar, Christine A Parlour, and Marius Zoican. Liquidity fragmentation on decentralized exchanges. *Available at SSRN 4267429*, 2022.

[221] Robert Leshner and Geoffrey Hayes. Compound: The money market protocol. *White Paper*, 2019.

[222] Lamport Leslie. The part-time parliament. *ACM Trans. on Computer Systems*, 16:133–169, 1998.

[223] Yudi Levi. Bancor's response to today's smart contract vulnerability. https://web.archive.org/web/20210525131534/https://blog.bancor.network/bancors-response-to-today-s-smart-contract-vulnerability-dc888c589fe4?gi=5e2d9c4ff877, Jun 2020.

[224] David A Levin and Yuval Peres. *Markov Chains and Mixing Times*, volume 107. American Mathematical Soc., 2017.

[225] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, 2014.

[226] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making {Geo-Replicated} systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[227] Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 359–372, 2018.

[228] Rujia Li, Xuanwei Hu, Qin Wang, Sisi Duan, and Qi Wang. Transaction fairness in blockchains, revisited. *Cryptology ePrint Archive*, 2023.

[229] Haoran Lin, Yajin Zhou, and Lei Wu. Operation-level concurrent transaction execution for blockchains. *arXiv preprint arXiv:2211.07911*, 2022.

[230] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. *ACM SIGOPS Operating Systems Review*, 25(5):226–238, 1991.

[231] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416, 2011.

[232] Ramon Llull and Jordi Gaya. *Ars notatoria*. Citema, 1978.

[233] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 80–96, New York, NY, USA, 2019. Association for Computing Machinery.

[234] James Lovejoy, Madars Virza, Cory Fields, Kevin Karwaski, Anders Brownworth, and Neha Narula. Hamilton: A high-performance transaction processor for central bank digital currencies. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 901–915, 2023.

[235] Tyler Lu and Craig Boutilier. Multi-winner social choice with incomplete preferences. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[236] Ananth Madhavan. Trading mechanisms in securities markets. *the Journal of Finance*, 47(2):607–641, 1992.

[237] Andrew Makhorin. Glpk (gnu linear programming kit). *http://www.gnu.org/s/glpk/glpk.html*, 2008.

[238] Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (mev) protection on a dag. *arXiv preprint arXiv:2208.00940*, 2022.

[239] Akaki Mamageishvili, Mahimna Kelkar, Jan Christoph Schlegel, and Edward W Felten. Buying time: Latency racing vs. bidding in fair transaction ordering. *arXiv preprint arXiv:2306.02179*, 2023.

[240] Viktor Manahov. Front-running scalping strategies and market manipulation: why does high-frequency trading need stricter regulation? *Financial Review*, 51(3):363–402, 2016.

[241] Fernando Martinelli and Nikolai Mushegian. Balancer whitepaper. Technical report, 9 2019. Accessed 2/4/2022.

[242] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32, 2015.

[243] Jason Milionis, Ciamac C Moallemi, and Tim Roughgarden. A myersonian framework for optimal liquidity provision in automated market makers. *arXiv preprint arXiv:2303.00208*, 2023.

[244] Jason Milionis, Ciamac C Moallemi, Tim Roughgarden, and Anthony Lee Zhang. Automated market making and loss-versus-rebalancing. *arXiv preprint arXiv:2208.06046*, 2022.

[245] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[246] T Mizuta and K Izumi. Investigation of frequent batch auctions using agent based model. In *JPX working paper*. Japan Excgange Group, 2016.

[247] Nicolas Mohnblatt. Sangria: a folding scheme for plonk. https://github.com/geometryresearch/technical_notes/blob/main/sangria_folding_plonk.pdf, 2023.

[248] Mojang Studios. Minecraft. https://www.minecraft.net/, 2009.

[249] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. http://bitcoin.org/bitcoin.pdf.

[250] EI Nenakov and ME Primak. One algorithm for finding solutions of the arrow-debreu model. *Kibernetica*, 3:127–128, 1983.

[251] Reserve Bank of Australia. Research project exploring use cases for cbdc. https://web.archive.org/web/20230301223652/https://www.rba.gov.au/media-releases/2023/mr-23-06.html, March 2023.

[252] Bank of England and HM Treasury. The digital pound: a new form of money for households and businesses? https://www.bankofengland.co.uk/-/media/boe/files/paper/2023/the-digital-pound-consultation-working-paper.pdf, February 2023.

[253] Board of Governors of the Federal Reserve System. Federal reserve announces that its new system for instant payments, the fednowő service, is now live. https://web.archive.org/web/20230721012415/https://www.federalreserve.gov/newsevents/pressreleases/other20230720a.htm, June 2023.

[254] Reserve Bank of India. Concept note on central bank digital currency. https://web.archive.org/web/20230331170938/https://www.rbi.org.in/Scripts/PublicationReportDetails.aspx?UrlPage=&ID=1218, October 2020.

[255] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[256] Working Group on E-CNY Research and Development of the People's Bank of China. Progress of research and development of E-CNY in china. http://www.pbc.gov.cn/en/3688110/3688172/4157443/4293696/2021071614584691871.pdf, Jul 2021. Accessed 12/14/2021.

[257] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[258] Abraham Othman, David M Pennock, Daniel M Reeves, and Tuomas Sandholm. A practical liquidity-sensitive automated market maker. *ACM Transactions on Economics and Computation (TEAC)*, 1(3):1–25, 2013.

[259] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092. USENIX Association, August 2020.

[260] Andreas Park. Conceptual flaws of decentralized automated market making. *Available at SSRN 3805750*, 2022.

[261] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. 2020.

[262] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[263] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[264] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

[265] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 198–214. IEEE, 2022.

[266] Sonbol Rahimpour and Majid Khabbazian. Spear: fast multi-path payment with redundancy. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 183–191, 2021.

[267] Geoffrey Ramseyer and Ashish Goel. Fair ordering via social choice theory. *arXiv preprint arXiv:2304.02730*, 2023.

[268] Geoffrey Ramseyer, Ashish Goel, and David Mazières. Liquidity in credit networks with constrained agents. In *Proceedings of The Web Conference 2020*, pages 2099–2108, 2020.

[269] Geoffrey Ramseyer, Ashish Goel, and David Mazières. SPEEDEX: A scalable, parallelizable, and economically efficient decentralized EXchange. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 849–875, Boston, MA, April 2023. USENIX Association.

[270] Geoffrey Ramseyer, Mohak Goyal, Ashish Goel, and David Mazières. Augmenting batch exchanges with constant function market makers. *arXiv preprint arXiv:2210.04929*, 2022.

[271] Geoffrey Ramseyer and David Mazières. Groundhog: Scaling smart contracting through commutative transaction semantics. 2023.

[272] Daniël Reijsbergen and Tien Tuan Anh Dinh. On exploiting transaction concurrency to speed up blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1044–1054. IEEE, 2020.

[273] Dan Robinson. Uniswap v3: The universal amm. https://web.archive.org/web/20210622195731/https://www.paradigm.xyz/2021/06/uniswap-v3-the-universal-amm/, Jun 2021.

[274] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability. *arXiv preprint arXiv:1906.08936*, 2019.

[275] Michael H Rothkopf, Aleksandar Pekeč, and Ronald M Harstad. Computationally manageable combinational auctions. *Management science*, 44(8):1131–1147, 1998.

[276] Tim Roughgarden. Tweet thread on cfmm research directions. https://web.archive.org/web/20221128200801/https://twitter.com/Tim_Roughgarden/status/1465095782533582858, Nov 2021. Accessed 08/03/2023.

[277] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 543–557, 2020.

[278] Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376*, 2019.

[279] Ferdinand Mongin Saussure. *Course in general linguistics*. Columbia University Press, [1916] 2011.

[280] Leopold Schabel. Reflections on solana's sept 14 outage. https://web.archive.org/web/20211104012332/https://jumpcrypto.com/reflections-on-the-sept-14-solana-outage/, Oct 2021. Accessed 12/7/2021.

[281] Jan Christoph Schlegel, Mateusz Kwaśnicki, and Akaki Mamageishvili. Axioms for constant function market makers. In *Proceedings of the 24th ACM Conference on Economics and Computation*, EC '23, page 1079, New York, NY, USA, 2023. Association for Computing Machinery.

[282] Noah Schmid, Christian Cachin, Orestis Alpos, and Giorgia Marson. Secure causal atomic broadcast, 2021.

[283] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[284] Robert A Schwartz. *The electronic call auction: Market mechanism and trading: Building a better stock market*, volume 7. Springer Science & Business Media, 2012.

[285] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*, 2018.

[286] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[287] Clara Shikhelman and Sergei Tikhomirov. Unjamming lightning: A systematic approach. *Cryptology ePrint Archive*, 2022.

[288] Alexander Spiegelman and Rati Gelashvili. Block-stm: How we execute over 160k transactions per second on the aptos blockchain. https://medium.com/aptoslabs/block-stm-how-we-execute-over-160k-transactions-per-second-on-the-aptos-blockchain-3b003657e4ba.

[289] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 1–17, 2021.

[290] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[291] NEAR Team. Near launches nightshade sharding, paving the way for mass adoption. https://web.archive.org/web/20221007081239/https://near.org/blog/near-launches-nightshade-sharding-paving-the-way-for-mass-adoption/, November 2021. Accessed 10/18/2022.

[292] SundaeSwap Team. Sundaeswap scalability. https://web.archive.org/web/20220523234648/https://sundaeswap.finance/posts/sundaeswap-scalability, November 2021.

[293] Tether. Tether: Fiat currencies on the bitcoin blockchain. https://web.archive.org/web/20211103054629/https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf. Accessed 12/14/2021.

[294] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[295] T Nicolaus Tideman. Independence of clones as a criterion for voting rules. *Social Choice and Welfare*, 4(3):185–206, 1987.

[296] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[297] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[298] Mohammad Amin Vafadar and Majid Khabbazian. Condorcet attack against fair transaction ordering. *arXiv preprint arXiv:2306.15743*, 2023.

[299] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.

[300] Mikhail Vladimirov and Dmitry Khovratovich. Erc20 api: An attack vector on the approve/transferfrom methods. https://web.archive.org/web/20221108114451/https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit.

[301] Fabian Vogelsteller and Vitalik Buterin. Eip 20: Erc-20 token standard. *Ethereum Improvement Proposals*, 20, 2015.

[302] Tom Walther. Multi-token batch auctions with uniform clearing prices - features and models. 3 2021.

[303] Gerui Wang, Shuo Wang, Vivek Bagaria, David Tse, and Pramod Viswanath. Prism removes consensus bottleneck for smart contracts. In *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 68–77. IEEE, 2020.

[304] Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. 2017.

[305] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pages 30–46, 2022.

[306] E Glen Weyl, Puja Ohlhaver, and Vitalik Buterin. Decentralized society: Finding web3's soul. *Available at SSRN 4105763*, 2022.

[307] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 21, 2016.

[308] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[309] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: a lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

[310] Alex Luoyuan Xiong, Binyi Chen, Zhenfei Zhang, Benedikt Bünz, Ben Fisch, Fernando Krell, and Philippe Camacho. Veri-zexe: Decentralized private computation with universal setup. *Cryptology ePrint Archive*, 2022.

[311] Anatoly Yakovenko. Sealevel: Parallel processing thousands of smart contracts. https://web.archive.org/web/20220124143042/https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192. Accessed 12/6/2021.

[312] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. *Whitepaper*, 2018.

[313] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.

[314] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, Renton, WA, April 2022. USENIX Association.

[315] Yinyu Ye. A path to the Arrow–Debreu competitive market equilibrium. *Mathematical Programming*, 111(1-2):315–348, 2008.

[316] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

[317] Wei Yu, Kan Luo, Yi Ding, Guang You, and Kai Hu. A parallel smart contract model. In *Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence*, pages 72–77, 2018.

[318] Eberhard Zeidler. *Nonlinear Functional Analysis and its Applications III: Variational Methods and Optimization*. Springer Science & Business Media, 1985.

[319] Eberhard Zeidler. *Applied Functional Analysis: Main Principles and their Applications*, volume 109. Springer Science & Business Media, 1995.

[320] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galinanes, and Bryan Ford. Flash freezing flash boys: Countering blockchain front-running. In *The Workshop on Decentralized Internet, Networks, Protocols, and Systems (DINPS)*, 2022.

[321] Jianting Zhang, Zicong Hong, Xiaoyu Qiu, Yufeng Zhan, Song Guo, and Wuhui Chen. Skychain: A deep reinforcement learning-empowered dynamic blockchain sharding system. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[322] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 633–649, 2020.

[323] 日本銀行決済機構局. 中央銀行デジタル通貨に関する実証実験「概念実証フェーズ 2」結果報告書. https://www.boj.or.jp/paym/digital/dig230417a.pdf, translated at https://www.boj.or.jp/en/paym/digital/dig230529a.pdf, May 2023.