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

Geoffrey Ramseyer
Stanford University

Ashish Goel
Stanford University

David Mazières
Stanford University

Abstract

SPEEDEX is a decentralized exchange (DEX) that lets participants securely trade assets without giving any single party undue control over the market. SPEEDEX offers several advantages over prior DEXes. 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 A to B always receives as good a price as trading through some third asset such as USD. Finally, it prevents 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 that is both asymptotically efficient and empirically practical that computes these valuations while exactly preserving a DEX'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 scheduled for deployment within one of the largest Layer-1 blockchains this year.

1 Introduction

Digital currencies are moving closer to mainstream adoption. Examples include central bank digital currencies (CBDCs) such as China's DC/EP [73], commercial efforts such as Diem [29], and many decentralized-blockchain-based stablecoins such as Tether [81], Dai [7], and USDC [13]. 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 interoperability platform where many digital currencies can efficiently coexist.

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 120 USD for 100 EUR, and might be matched against another user who previously offered to trade 100 EUR for 120 USD. A typical exchange maintains "orderbooks" of all of the open trade offers.

The ideal digital currency exchange should

- 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.

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 [39] for private blockchains or synchronous mining [72] 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, one might spy an incoming sell offer, and in response, send a trade that buys and

immediately resells an asset at a higher price [34, 67]. On a blockchain, where a block of trades either is finalized entirely or not at all, this front-running can be made risk-free. More generally, some users form special connections with blockchain node operators to gain preferential treatment for their transactions [49]. 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 [77].

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 artificially limited by the rate at which hardware manufacturers design faster CPUs. An ideal DEX should handle higher transaction rates simply by using more compute hardware.

Unfortunately, conventional wisdom holds that DEXes cannot 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 [84], automated market-makers [19], transaction rollup systems [11, 15], and sharded blockchains [4] or side-chains [75]. 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 buying 1 EUR might consume an offer priced at 1.20 USD, leaving the second to pay 1.22 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. Unfortunately, straightforward parallel execution would make the order of transactions dependent on non-deterministic inter-thread scheduling.

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 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 on a 48-core machine (Figure 5). 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 all transactions in a block commute: the block's result is identical regardless of the order in which trades are executed, which in turn allows efficient parallelization [44].

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 whole block of limit orders as one unified batch, in which, for example, every 1 EUR sold to buy USD receives exactly 1.21 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 (in nonempty markets) for any batch of trades—SPEEDEX operators cannot choose these rates strategically. 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 limit order's minimum 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 admits no risk-free front-running. 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.

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 effectively parallelized, allowing SPEEDEX to scale to support transaction volumes far beyond those seen today. Transactions within a block commute with each other precisely because asset transfers 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.

1.2 SPEEDEX Overview

SPEEDEX is not a blockchain itself, but rather, is a DEX component that can be integrated into any blockchain. A copy of the SPEEDEX module should run inside every blockchain replica. It does not depend on any specific property of a consensus protocol, but automatically benefits from throughput advances in consensus and transaction dissemination (such as [50]). 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 have implemented SPEEDEX within a blockchain running the HotStuff consensus protocol [89]; this implementation provides the performance measurements in this paper. We have also implemented SPEEDEX as a component of the Stellar blockchain, one of the largest Layer-1 cryptocurrency platforms [48]. This blockchain will deploy SPEEDEX in a Layer-1 protocol upgrade in 2022. An additional blockchain company anticipates its own future 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); 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 [46]) that can, in practice, efficiently compute approximate batch clearing prices. We then explicitly correct approximation error with a follow-up linear program.

On the systems design side, to implement this exchange, we design a set of natural transaction semantics that admit concurrent transaction processing and a set of data structures designed for concurrent manipulation and for efficiently answering queries about the exchange state from the price computation algorithm.

2 System Architecture

SPEEDEX is an asset exchange implemented as a replicated state machine in a blockchain architecture (Figure 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. 1, 1) among block *producers*. At each round, one or more producers propose candidate blocks extending the blockchain history (Fig. 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. 1, 3). SPEEDEX is suitable for integration into a variety of blockchains, but benefits from a consensus layer with relatively low latency

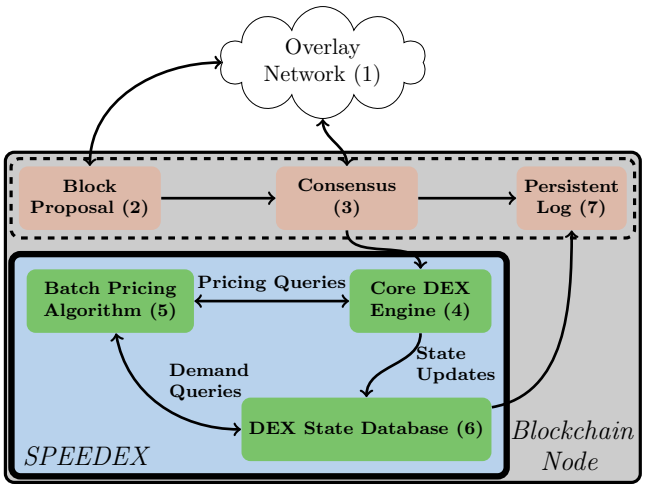

Figure 1: Diagram of SPEEDEX module (4, 5, 6) integrated into one blockchain node.

(on the order of seconds), such as BA⋆ [60], SCP [69], or HotStuff [89].

The implementation evaluated here uses HotStuff [89], while the deployment in the Stellar blockchain will use Stellar's existing (non-Hotstuff) consensus protocol.

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 asset that is divisible and fungible can integrate smoothly 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 1.20 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 0.80 EUR/USD.

What makes SPEEDEX different from existing DEXes is the manner in which it processes incoming trade offers. Traditional DEXes process trades sequentially, implicitly computing a matching between trade offers. SPEEDEX, by contrast, is explicitly designed to operate in a blockchain context.

In a blockchain, all of the transactions in a block are appended at the same clock time. As such, there is no reason a priori why the DEX should pick one ordering of a block's transactions 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. 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. 1, 6). This database records orderbooks and account balances, and is periodically written to the persistent log (Fig 1, 7).

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. Traditionally, 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. Note that every trade occurs at a slightly different exchange rate.

Instead, to make trades commutative, SPEEDEX computes in every block a *valuation* p_A for every asset A. The units of p_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 A for asset B occurs at a price of p_A/p_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 auctioneers 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.

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

Theorem 1 follows from general market equilibrium theory [51] (§A.3).

Concretely, whenever the core SPEEDEX engine (Fig 1, 4) receives a newly finalized block, one of its first actions is to query an algorithm that computes clearing valuations (Fig 1, 5). It then uses the output of this algorithm to compute the modifications to the exchange state (Fig 1, 6).

Because valuations that can clear the market always exist for any set of limit orders, there is no possibility of 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, therefore, is to surface information corresponding to a fundamental mathematical property of a batch of trade offers.

However, computing clearing valuations exactly appears computationally infeasible (for example, even the number of bits required to write down exact clearing prices could be extremely large [51]). SPEEDEX thus computes approximate clearing prices.

Approximation error comes in two forms (§B): first, like most exchanges, SPEEDEX charges a small commission on every trade. Second, the amount of trade volume in one round is approximately optimal; at nonexact clearing prices, there is an imbalance between the amount of an asset sold to the auctioneer and the amount bought from it.

The commission permits SPEEDEX to operate when these imbalances are small. SPEEDEX corrects any remaining imbalances by occasionally not executing some offers with in-the-money limit prices that are very close to the auctioneer's exchange rate. Note that the commission is not needed to cover operational costs, and could be sent to any party in the SPEEDEX ecosystem.

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 Development Foundation plans to eliminate the fee (for a tradeoff of possibly increasing approximation error of the second type), and return the accumulated rounding error to asset issuers.

2.2 Design Properties

Computational Scalability SPEEDEX's commutative semantics allow effective parallization of DEX operation. Because transactions within a block are not semantically ordered, DEX state is replicable (after applying all transactions in a block) no matter the ordering in which they are applied. This replicability is, of course, required in replicated state machine.

As such, even though thread scheduling on multicore CPUs is nondeterministic, SPEEDEX can use every available CPU core to apply transactions without any significant inter-thread coordination. In fact, almost all coordination occurs via hardware-level atomics (e.g., atomic add on 64-bit integers) without spinlocks.

Note that the commutative semantics of SPEEDEX support generic payment operations in addition to DEX trading. SPEEDEX stores balances within accounts, and not in UTXOs, and therefore demonstrates that contrary to popular belief (e.g. as discussed in [70]), horizontal scalability and an account-based data store are not incompatible.

No front running ("MEV" reduction) 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 [49]. 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) [49] phenomenon, in which block producers reorder transactions within a block for their own profit (or in exchange for bribes). By eliminating the ordering of transactions within a block, SPEEDEX eliminates this large source of MEV (nodes, however, might still try to delay transactions).

No (internal) arbitrage and no central reserve currency An agent selling asset A in exchange for asset B will see a price of p_A/p_B. An agent trading A for B via some intermediary asset C will see exactly the same price, as $\frac{p_A}{p_C} \cdot \frac{p_C}{p_B} = \frac{p_A}{p_B}$. 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, and the resulting liquidity fragmentation, further complicates the problem on existing blockchains. Of course, there may be arbitrage between SPEEDEX and external markets.

One downside of a batching-based trading system is the latency between trade submission and execution (relative to that of trading immediately against a centralized orderbook). However, the design of blockchains inherently introduces latency between transaction submission and finalization. In this context, SPEEDEX's design introduces no additional latency.

3 Commutative DEX Semantics

To process a finalized 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 trade offers, and compute available account balances after funds are committed to offers or transferred between accounts.
2. Compute approximate clearing prices and approximation correction metadata.
3. Iterate over every trade offer, possibly executing the offer based on the computed clearing 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 of 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.

Because transactions are not ordered with respect to each other, SPEEDEX cannot resolve a double spend by failing the "second" transaction. Instead, SPEEDEX necessitates that after processing all transactions in a block, all accounts have a nonnegative amount of "unlocked" capital of each asset type (where an open offer locks the offered amount of an asset for the duration of its lifetime). Block producers are responsible for ensuring this property holds, and proposals that do not satisfy this property are ignored. This design requires passing information from the SPEEDEX database (Fig 1, 6) to the proposal module (Fig 1, 2). By contrast, the Stellar Development Foundation plans an approach that fully separates consensus from block execution. A preprocessing step in the Stellar blockchain identifies all accounts active in a block that attempt to spend or lock more of an asset than they have, and prunes out some of the transactions from these accounts.

The core remaining technical challenge is the batch price computation (Fig 1, 5).

4 Price Computation

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 §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 many trades as possible. (Otherwise, the constraints could be vacuously met by a DEX that never actually trades.)

Furthermore, this algorithm must be efficient; SPEEDEX needs to run this algorithm on every block of trades (i.e. once every few seconds). And finally, SPEEDEX should minimize the number of offers that trade partially (asset quantities are stored as integers; every fractional trade necessarily accumulates some rounding error).

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). Our algorithms are based on algorithms from this literature (an iterative process known as Tâtonnement [46]).

However, the runtimes of these algorithms scale extremely 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 [46] permits equilibria to manufacture new assets and to take money from a user without giving anything in return).

We develop a novel algorithm for computing equilibria that runs efficiently in practice (§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 below the batch exchange rate.

§5 describes how we use the formal structure of the types of trades in SPEEDEX to design an algorithm that can run effectively in practice on a wide range of market conditions and trading patterns (and is formally, asymptotically faster; the runtime of an iteration is logarithmic in the number of open offers).

We then explicitly correct for the approximation error in Tâtonnement with a linear program (§D). 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, the output of our batch pricing algorithm consists of the following:

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

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

The Stellar blockchain plans to run SPEEDEX with no transaction commission, a choice which makes the linear program into an instance of a "maximum circulation" problem, a subclass of linear programs that can be efficiently solved exactly (without any floating-point arithmetic. See §D).

5 Price Computation: Tâtonnement

SPEEDEX's price computation algorithm is based on an iterative algorithm known as Tâtonnement [46]. The algorithm starts at some (arbitrary) set of initial prices, and iteratively refines them until the prices meet a stopping criterion.

Each iteration of Tâtonnement starts with a "demand query." The "net demand" of an offer is the net trading of the offer (with the "auctioneer") in response to a set of prices.

Example 1. *Suppose that one limit order offers to sell 100 USD in exchange for EUR, and demands a minimum of* 0.8 *EUR per USD (that is, 1 EUR trades for 1.25 USD).*

If the candidate prices of USD and EUR are such that $\alpha = \frac{p_{USD}}{p_{EUR}} < 0.8$*, then the limit order would like to trade, and so its demand is* $(-100\ USD, \alpha * 100\ USD)$*. Otherwise, its demand is* $(0\ USD, 0\ EUR)$*.*

The net demand of a set of offers is the sum of the demands of every individual offer. The goal of Tâtonnement is to find prices such that the net demand of the entire set of open offers is as close to 0 (for every asset) as possible.

Iterative Price Adjustment If the net demand of an asset is positive, then more units of the asset are demanded from the "auctioneer" than are supplied to it (so the "auctioneer" has a deficit). In response, the "auctioneer" raises the price of the asset. Otherwise, the "auctioneer" has a surplus, so it lowers the price of the asset (we give the precise update formula in §C).

Tâtonnement repeats this iterative process until either it times out or the current set of prices is sufficiently close to the unique market clearing prices. Specifically, Tâtonnement iterates until it has a set of prices such that, if the "auctioneer" charges a transaction 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 prics far below the "auctioneer"'s exchange rate strongly prefers trading to not trading.

5.1 Efficient Demand Queries

Implemented naively, Tâtonnement's demand queries would consist of a loop over every open exchange offer. This is impossibly expensive, even if this 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 naive loop appears to be required for the (more general) problem instances studied in the theoretical literature (even for markets with so-called "linear utilities"). However, all of the offers traded in SPEEDEX have a particular form. They 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.

SPEEDEX can therefore compute a demand query with a sequence of binary searches on these sorted lists (§G). Individual binary searches can run on separate CPU cores.

In general, the number of open offers (say, M) on an exchange is vastly higher than the number of assets traded (say, N). Our experiments in §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 massive.

Done naively, the sorting step in each block would be prohibitively expensive. However, careful design of our orderbook data structures drives the marginal cost of this sorting to near 0 (§8.5).

5.2 Multiple Tâtonnement Instances

We make a number of other adjustments to Tâtonnement, which we outline in §C, that help Tâtonnement respond well to a wide variety of problem instances. Some of these are specified by various control parameters (such as how quickly one should adjust the candidate prices); rather than pick any 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 "lost utility", see §6.2).

SPEEDEX therefore includes the output of Tâtonnement in the headers of proposed blocks (§8.3).

The Stellar blockchain's implementation runs only one instance of Tâtonnement for a fixed number of rounds (at the cost of possibly less accurate Tâtonnement output).

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, §C.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.

The biggest factor limiting the number of assets is the follow-up linear program, whose runtime increases dramatically beyond 60-80 assets. (The tests here use 50 assets.) This increase relates to the quadratic increase in the size of the program. A deployment could avoid this runtime increase by taking 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 as such are primarily traded against a single currency. We formally show in §E that the price computation problem can be cleanly decomposed between the core "pricing" currencies and the external stocks. After running Tâtonnement on the core currencies, the the stocks can be priced as a post-processing step. This lets SPEEDEX to support typical real world transaction patterns

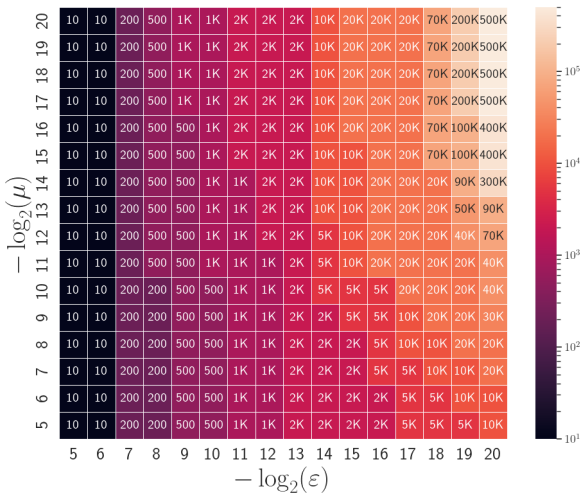

Figure 2: Minimum number of offers needed for Tâtonnement to run in under 0.25 seconds (Smaller is better. Times averaged over 5 runs). The x axis denotes offer behavior approximation quality (μ), and the y axis denotes the commission (ε).

with an arbitrary number of assets and a small number of core pricing currencies.

6.1 Approximation Accuracy and Orderbook Size

We find that Tâtonnement converges more quickly as the number of open offers *increases*. Specifically, Tâtonnement converges fastest when small price changes do not cause comparatively large changes in overall net demand.

However, the behavior of a single offer is a discontinuous function (of prices); an order does not trade at all below its minimum price and trades in full above it.

There are two factors that mitigate the difficulty of these "jump discontinuities." First, Tâtonnement approximates optimal offer behavior by a smoother function (§B). Smaller μ means a closer approximation, meaning the offer behaves more like the original discontinuous function. 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 actually converges more quickly when more offers are present on the exchange. A real-world deployment might tighten the approximation bounds as network activity increases.

Figure 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 §7).

For comparison, BinanceDex [1] charges a fee of either $0.1\% \approx 2^{-10}$ or $0.04\% \approx 2^{-11.3}$. Uniswap [19,20] can charge 1%, 0.3%, or 0.05% (i.e., $\sim 2^{-6.6}$, $\sim 2^{-8.4}$, and $\sim 2^{-11}$, respectively). Coinbase charges between 0.5% and 4% depending

on the transaction [3] (approximately $2^{-7.6}$ to $2^{-4.6}$).

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. If the number of open offers is small, a deployment could also invoke a different solving strategy (§F).

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 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 A (and buys asset B) with probability proportional to the relative volume of asset A (and asset B, conditioned on $B \neq A$) 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 $\sim 30,000$ (in comparison, for scalability, we test 500,000-transaction blocks in §7).

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

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, Tâtonnement continues to refine the price of every asset in every iteration. This is why Tâtonnement actually gives more accurate results in the blocks it found challenging. If anything, 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 Figure 2). Tâtonnement also adjusts its price adjustment rule in response to recent market conditions (§C.1), a tactic which is less effective on extremely volatile assets.

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

7 Evaluation: Scalability

We ran SPEEDEX on four c5d.metal instances in an Amazon Web Services datacenter. Each machine has two 24-core Intel Xeon Platinum 8275CL CPUs, running at 3.00 Ghz with hyperthreading enabled, 192GB of memory, and 4 900GB NVMe drives connected in a RAID0 configuration. We use the XFS filesystem [80]. These experiments use the HotStuff consensus protocol [89], 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 sell orders priced below 0.999 times the auctioneer's price. The initial account database contains 10 million accounts. Tâtonnement never timed out.

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, akin to a geometric Brownian motion, after every set. Transaction sets include a mix of new offers, cancellations of previous offers, payments between accounts, and new account creations.

Each set is then split into four pieces, with one piece given to each replica. During an experiment, replicas load these sets sequentially and broadcast each set to every other replica. Each replica adds received transactions to its memory pool.

Replicas propose blocks of roughly 500,000 transactions.

Figure 3: Time to propose a block with varying numbers of worker threads (48 physical CPU cores), plotted over the number of open offers on the exchange.

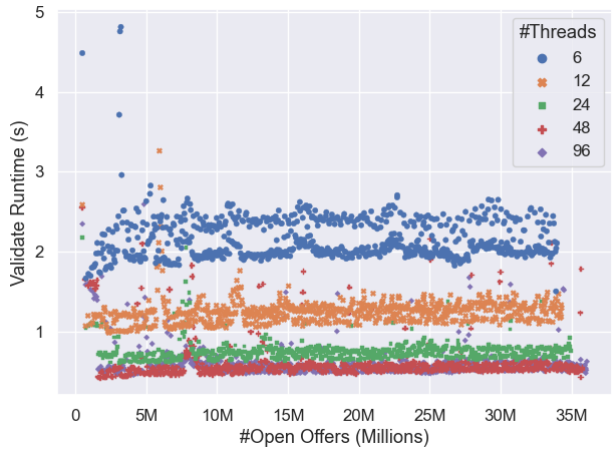

Figure 4: Time to validate a block with varying numbers of worker threads (48 physical CPU cores), plotted over the number of open offers on the exchange (measurements from one replica).

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.

Performance Measurements Figures 3 and 4 plot the runtime to propose and validate SPEEDEX blocks, respectively, with varying numbers of worker threads. The x-axes denote the number of open offers on the exchange. Other than the number of transactions in a block, the number of open offers on SPEEDEX is the biggest factor influencing runtime of block proposal or validation. These graphs demonstrate that SPEEDEX does benefit from access to more CPUs (although the benefit of hyperthreads appears marginal). To highlight the scalability of SPEEDEX, the measurements plotted here do not include signature verification, which is easily parallelizable and would therefore bias the measurements. For comparison, the runtimes of block proposal and validation with a few million open trade offers, using only a single worker thread, are approximately 12-13 seconds and 10 seconds, respectively.

Most computation is done by a set of worker threads. To demonstrate scalability, we measure the runtime of SPEEDEX nodes with varying numbers of worker threads. This makes the performance of the experiments using fewer threads look better than it should be. Our implementation is designed around having access to many cpu cores, so running SPEEDEX unmodified on a system with few CPU cores would not be a fair comparison.

The increase in runtime associated with an increase in the number of open offers stems from a Tâtonnement optimization (the precomputation outlined in §8.2). The one part of SPEEDEX that cannot be arbitrarily parallelized is Tâtonnement, and so we design our implementation towards making Tâtonnement as fast as possible. An implementation might skip this work in some parameter regimes, and nodes always skip this work when validating a block.

The runtime variation when validating a block with a small number of threads likely relates to variation in thread scheduling and the interaction between scheduling and the system's NUMA architecture.

Finally, Figure 5 plots the overall transaction rate of SPEEDEX (the number of transactions in a block, divided by the difference in clock time between sequential invocations of the block proposal method). Figure 5 includes signature verification time and a checkpointing procedure.

Every five blocks, the exchange logs its state to disk, through LMDB [43]. This generates a sustained write workload of 500 to 700 MB/s. The occasionally slow runtimes of Figure 5 are the result of waiting for this logging to finish (see §I.2). Running SPEEDEX with fewer threads does not help the log work faster, but does give it more time.

Conclusions To reiterate, SPEEDEX achieves these transaction throughput numbers—more than 200,000 per second (150,000 if one is conservative about waiting for data persistence)—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.

The commutative transaction semantics also benefit payment workloads. Our implementation peaks at 350,000 transactions per second when processing blocks of only payments (although again, our log implementation does not keep up

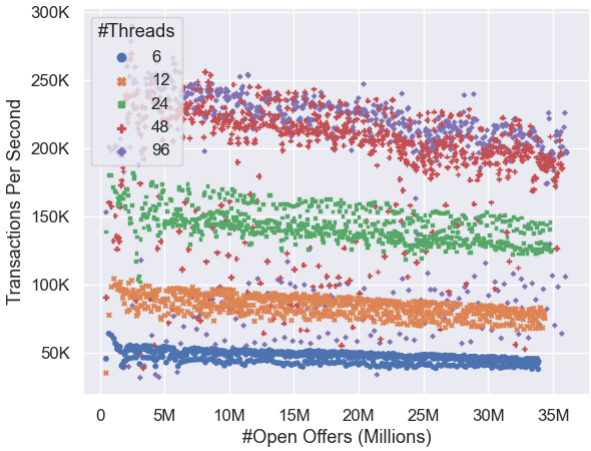

Figure 5: Transactions per second on SPEEDEX with varying numbers of worker threads (48 physical CPU cores), plotted over the number of open offers on the exchange.

with that transaction rate).

7.1 Comparison with Existing DEXes

Many current blockchain or DEX scaling projects (such as [8, 9, 14, 15, 64]) move transaction execution off-chain. These solutions require explicit user interaction to move funds in and out of the system, are throughput-limited by the speed of a single CPU core, and lose some of the censorship-resistance and auditability benefits of a decentralized system.

The Stellar blockchain implements an orderbook-based DEX directly within its server daemon. Precise timing measurements of its (carefully optimized) production services show that, calculated optimistically, its implementation could handle $\sim$ 4000 DEX trades per second.

Wang et al. [83] measure the raw transaction throughput of the Ethereum Virtual Machine; with 10,000 open accounts, the EVM processed token transfer transactions at a rate of 13 thousand per second. Complex transactions run substantially more slowly. The Loopring exchange, an L2 rollup built on Ethereum, claims a maximum trade rate of $\sim$ 2000 per second [18]. This number is based off of Ethereum's gas limit [17], which is set in response to the real computational cost of serial transaction execution [21, 38, 40, 74].

§F describes some alternative price computation strategies and their observed scalability.

8 Implementation Details

The standalone SPEEDEX evaluated in §6 and §7 is a blockchain using HotStuff [89] 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 publicly available open source and consists of $\sim$40,000 lines of C++20. It relies on Intel's TBB library [6] for much of the parallel work coordination, the GNU Linear Programming Kit [71] as the linear program solver, and LMDB [43] to manage data persistence and 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 remainder of this section outlines some additional design choices built into SPEEDEX. Additional design choices in §I. All optimizations (save §8.1) are implemented within the evaluated system.

8.1 Blockchain Integration

An existing blockchain with its own (non-commutative) semantics can integrate SPEEDEX by splitting block execution into phases: first applying all the SPEEDEX transactions (in parallel), then applying the legacy transactions (sequentially). SPEEDEX's scalability lets a blockchain charge users only a marginal gas fee for simple payments and trades.

A proof-of-stake integration of SPEEDEX could choose to penalize a faulty block producer.

SPEEDEX's economic properties are desirable independent of scalability. The Stellar blockchain's first SPEEDEX release will use two-phase blocks, but the SPEEDEX phase is still implemented sequentially. As a result, the implementation is much simpler (adding only $\sim$5,000 lines to the main server daemon) and the primary benefits are economic. Once the system supports commutative SPEEDEX transactions, engineers can parallelize the implementation with no further protocol updates.

8.2 Caches and Tâtonnement

Most of Tâtonnement's runtime is spent computing demand queries. Each of these queries consists of several binary searches over large lists, so the runtime depends heavily on memory latency and cache performance.

Furthermore, especially 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 directly using the offer database in these queries, we precompute (in one pass) for each asset pair a list that records, for each unique price, the total amount of an asset available for sale below the current price (§G). Laying out all of the information for Tâtonnement contiguously in memory improves the cache performance of each aggregate demand query.

We further accelerate Tâtonnement by executing the binary searches in parallel. One primary thread computes price updates and wakes helper threads for demand queries. However, each round of Tâtonnement is extremely fast even on one thread; for example, with 50 assets and millions of offers to trade, one round requires 400 to 600 microseconds, on aver-

age. 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 50 asset tests of §6, we see minimal benefit beyond 4-6 helper threads, but this is enough to bring the average runtime per round down to 50 to 150 microseconds.

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

We accelerate the rest of a Tâtonnement round by using fixed-point arithmetic, rather than floating-point. This avoids (nonassociative) accumulation of floating-point error.

8.3 Block Header Design

Block producers include Tâtonnement and linear program output in a block proposal. Checking the correctness of asset valuations is much faster than running Tâtonnement. This choice also permits nondeterminism in Tâtonnement, which we use to run multiple Tâtonnement instances in parallel with different operational parameters (§5.2).

Block headers also include, for every pair of assets, the trie key of the offer with the highest minimum price that trades in that block. Validators can compare the trie key of a newly created offer with this marginal key and know immediately whether or not the offer executes.

The Stellar Development Foundation will not include these optimizations. Every node will run one Tâtonnement instance with a fixed set of operational parameters (and thus Tâtonnement will run deterministically). This choice lets Stellar fully separate consensus from SPEEDEX.

8.4 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 limits sequence numbers to increase by at most 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 Development Foundation has chosen to require strictly consecutive sequence numbers, mostly for backwards compatibility.

8.5 Fast Offer Sorting

The running times of §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.

To minimize contention, each thread privately accumulates tries recording newly created offers. We then gather these tries into sets grouped by trading pair and merge the sets in parallel.

8.6 Fast Merkle-Patricia Tries

SPEEDEX concurrently manipulates tries frequently, making it important to minimize memory contention. Our tries use a fan-out of 16 and hash nodes with the 32-byte BLAKE2b cryptographic hash [30]. Both the layout of trie nodes and the work partitioning are designed to avoid having multiple threads writing to the same cache line.

When applying transactions, each thread builds an ephemeral local trie of the accounts it has modified, with the modifying transactions at the leaves. At the end of transaction processing, we merge these tries by re-dividing them by prefix range, so that a given thread merges the same range from all tries. (Once the ranges are individually merged, it is trivial to merge a set of tries with disjoint key ranges.)

Ephemeral trie nodes fit within a single 64-byte cache line. We allocate the 16 children of an ephemeral trie node contiguously, requiring the parent to store only one base pointer plus a bitmap of the children in use. Memory allocation itself is optimized using thread-local arenas.

The ephemeral trie uses the same key space as the main account state trie (account ID), which lets us use the ephemeral trie to efficiently divide work on the account state trie.

Because the leaves of the ephemeral trie store transactions, building this trie implicitly sorts transactions by account ID, which reduces cache contention during validation when the list is partitioned into contiguous segments. (This is not enforced by validators, so a faulty block producer could slightly reduce performance by issuing otherwise valid but unsorted blocks.)

We optimize trie node contents to speed up operations. Most key comparisons are performed using 64-bit integers, rather than one byte at a time. Each trie node stores the number of leaves below it, so as to facilitate distributing work evenly among threads. Tries that store offers also store the number of canceled offers below each node and the total amount of the asset offered for sale below the node. Offers can be canceled via hardware-level atomics—the actual work of removing the canceled offer from a trie is deferred until after order processing to avoid synchronization overhead.

9 Extensions

Many automated market makers, such as those based on Uniswap's constant product rule [19], integrate naturally into SPEEDEX and do not require substantive changes to Tâtonnement [25]. The Stellar Development Foundation will

integrate constant-product market makers when it deploys SPEEDEX.

SPEEDEX does not implement offers to buy a fixed amount of an asset (e.g. buy at most 100 USD, spending as few EUR as possible). These offers actually make the price computation problem PPAD-hard, a complexity class that is widely conjectured to be intractable for polynomial-time algorithms (see §H).

It is still possible that Tâtonnement could handle a small number of buy offers. Buy offers also admit the same logarithmic transformation as in §5.1. Alternatively, one could compute prices only using sell offers, and then integrate buy offers in the linear programming step.

10 Related Work

10.1 Blockchain Scaling and Concurrency

Chen et al. [42] argue that execution of the Ethereum Virtual Machine is a bottleneck for Ethereum's transaction throughput, and achieve a roughly 6x end-to-end performance improvement by speculatively executing smart contracts. Gelashvili et al. [59] optimistically execute transactions in batches.

Solana [87] scales its runtime by executing non-conflicting transactions in parallel [86]. This increases throughput of the whole blockchain, but does not help scale a single contract. A recent network outage was caused in part by an overwhelming number of transactions trading on a single orderbook on Serum [10], Solana's on-chain DEX [79].

Project Hamilton [70] develops a protoype CBDC payments platform. They also find that totally-ordered transaction semantics in a replicated state machine quickly become a bottleneck for system performance. Unlike SPEEDEX, which stores asset balances in accounts, this system forces users to use the more restrictive "unspent transaction output" (UTXO) data model to achieve some parallelism.

Other projects move transaction execution off-chain, into so-called "Layer-2" networks. The Lightning network [76] routes payments through a network of bilateral channels, where each channel allows two parties to securely transact at a high rate.

Systems such as Plasma [75] extend the Layer-2 channel model. Users lock funds within a root contract on a main blockchain, then send transactions to aggregators. The model has many variants [8, 9, 14, 17, 64], each with different capabilities, performance, interoperability, and security properties, but common to these approaches is the idea of scaling overall system throughput by moving transaction execution off-chain. System security requires fraud-proof mechanisms for identifying malicious channel operators and (in the case of optimistic rollups) some requirements that users remain online.

Some blockchain designs [4, 22, 85, 91] split state into semi-independent shards. Cross-shard transactions are more complicated (and expensive) than single-shard transactions.

Saraph and Herlihy [78] argue that optimistic concurrency control could have historically made the EVM 2 to 8 times faster. They find that a few contracts, such as token contracts, are responsible for a large fraction of concurrency conflicts. Dickerson et al. [54] allow concurrency in smart contracts via software transactional memory and inclusion in block headers of conflict resolution information.

Hyperledger Fabric [22] concurrently executes transactions in isolation, later invalidating conflicting pairs.

Anjana et al. [24] record ordering dependencies in block headers. Yu et al. [90] identify conflicting transactions prior to execution. Bartoletti et al. [31] build a framework for statically identifying commuting smart contract calls.

Li et al. [68] build a distributed database that allows some transactions to be tagged as commutative (and reorderable).

Our approach is inspired by Clements et al. [44], who improve performance in the Linux kernel through commutative syscall semantics.

10.2 (Distributed) Exchanges

Some blockchains, such as Stellar [12] and BinanceDex [1], provide a built-in DEX mechanism. Serum [10] built a limit-order DEX as a smart contract on Solana.

Automated market makers, such as Uniswap [19] or Bancor [61], are smart contracts that trade with users directly. Their exchange rates are a function of their currency reserves [23]. Uniswap allows trading between any two asset pairs, while Bancor requires trades to pass through an intermediate reserve currency.

0x [84] provides an interface for atomic trades between untrusted parties, and acts as a building block for off-chain exchanges. An old version of Loopring [15] gave a similar interface but matched offers in cycles. StarkEx [11, 32] provides cryptographic tools that let a centralized exchange match trades off-chain and prove operational correctness.

To combat front-running, Clockwork [45] uses timelock puzzles to let an exchange commit to processing an offer before it sees the offer's contents. Zhang et al. [92] and Kelkar et al. [65] limit the power of node to choose a transaction ordering. Chainlink plans to use Kelkar et al.'s algorithm as a "Fair Sequencing Service" for other blockchain systems [36].

The Binance DEX computes per-asset-pair exchange rates. Newly created offers trade at that ratio, while pre-existing offers trade at their limit prices. The exchange currently handles 10–30 operations per second [2].

The Gnosis exchange [5] clears offers in batches of at most 100 [16] by solving a mixed-integer programming problem. Solvers compete to produce the best solution.

Budish et al. [37] argue that (centralized) exchanges should process orders in batches to combat automated arbitrage.

10.3 Price Computation

The core algorithms of this work deal with the special case of the Arrow-Debreu exchange market [28] where every agent has a linear utility function. Equilibria can be computed ap-

proximately in these markets using combinatorial algorithms such as those of Jain et al. [63] and Devanur et al. [52] and exactly via the ellipsoid method and simultaneous diophantine approximation [62]. Duan et al. [56] construct an exact combinatorial algorithm, which Garg et al. [58] extend to build a combinatorial algorithm with strongly-polynomial running time. Ye [88] describes a path-following interior point method, and Devanur et al. [51] construct a convex program.

Our algorithms are based on an interative process known as Tâtonnement [27]. Codenotti et al. [46, 47] show that this process converges to an approximate equilibrium in polynomial time.

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 different 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 directly trade 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 available open-source at https://github.com/scslab/speedex, and is scheduled for deployment in the Stellar blockchain in 2022.

Acknowledgements

This research was supported by the Stanford Future of Digital Currency Initiative, the Stanford Center for Blockchain Research, the Office of Naval Research (ONR N00014-19-1-2268), and the Army Research Office.

The authors also wish to thank CloudLab [57] and the Stellar Development Foundation for providing compute resources for our experiments.

References

[1] Binance chain docs - match engine. https://docs.binance.org/match.html. Accessed 03/14/2021.

[2] Binance chain explorer. https://explorer.binance.org/. Accessed 03/14/2021.

[3] Coinbase pricing and fees disclosures. https://help.coinbase.com/en/coinbase/trading-and-funding/pricing-and-fees/fees. Accessed 04/10/2021.

[4] Eth2 shard chains. https://ethereum.org/en/eth2/shard-chains/. Accessed 03/11/2021.

[5] 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 and https://github.com/gnosis/dex-research/blob/master/BatchAuctionOptimization/batchauctions.pdf.

[6] Intel oneapi threading building blocks. "https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html". Accessed 5/6/2021.

[7] The maker protocol: Makerdao's multi-collateral dai (mcd) system. https://makerdao.com/en/whitepaper/. Accessed 12/14/2021.

[8] Optimistic rollups. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic$_{r}$ollups/. Accessed 03/11/2021.

[9] Polygon lightpaper: Ethereum's internet of blockchains. https://polygon.technology/lightpaper-polygon.pdf. Accessed 12/6/2021.

[10] Serum: Faster, cheaper, and more powerful defi. https://www.projectserum.com/. Accessed 12/6/2021.

[11] Starkex. https://starkware.co/product/starkex/.

[12] Stellar. https://www.stellar.org/.

[13] USDC: the world's leading digital dollar stablecoin. https://www.circle.com/en/usdc. Accessed 12/14/2021.

[14] Zk rollups. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/. Accessed 03/11/2021.

[15] Loopring: A decentralized token exchange protocol. September 2018.

[16] Gpv2 objective criterion. https://forum.gnosis.io/t/gpv2-objective-criterion/1254, April 2021. Accessed 04/30/2021.

[17] Loopring 3 design doc. https://web.archive.org/web/20220411224154/https://github.com/Loopring/protocols/blob/master/packages/loopring$_v$3/DESIGN.md#results, 2021.

[18] Loopring protocol. https://web.archive.org/web/20220409050852/https://loopring.org/#/protocol, April 2022.

[19] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020.

[20] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. Technical report, Tech. rep., Uniswap, 2021.

[21] 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.

[22] 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.

[23] Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *Cryptoeconomic Systems Journal*, 2019.

[24] 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.

[25] Anonymized. Batch exchanges with constant function market makers: Axioms, equilibria, and computation. Unpublished Manuscript.

[26] Larry Armijo. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, 16(1):1–3, 1966.

[27] 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.

[28] Kenneth J Arrow and Gerard Debreu. Existence of an equilibrium for a competitive economy. *Econometrica: Journal of the Econometric Society*, pages 265–290, 1954.

[29] Libra Association. Diem white paper v2.0. https://www.diem.com/en-us/white-paper/, Apr 2020. Accessed 12/14/2021.

[30] Jean-Philippe Aumasson and Markku-Juhani O Saarinen. The blake2 cryptographic hash and message authentication code (mac). *RFC 7693*, 2015.

[31] 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.

[32] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. 2018.

[33] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of computational Physics*, 182(2):418–477, 2002.

[34] 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.

[35] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[36] 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://research.chain.link/whitepaper-v2.pdf, 2021. Accessed 12/14/2021.

[37] 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.

[38] 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$_q$uick$_e$xplanation$_o$f$_w$hat$_t$he$_p$oint$_o$f$_t$he$_e$ip/, April 2021.

[39] 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.

[40] 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.

[41] Xi Chen, Dimitris Paparas, and Mihalis Yannakakis. The complexity of non-monotone markets. *Journal of the ACM (JACM)*, 64(3):1–56, 2017.

[42] 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.

[43] Howard Chu and Symas Corporation. Lightning memory-mapped database manager (lmdb). http://www.lmdb.tech/doc/. Accessed 04/29/2021.

[44] 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.

[45] Dan Cline, Thaddeus Dryja, and Neha Narula. Clockwork: An exchange protocol for proofs of non front-running.

[46] 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.

[47] Bruno Codenotti, Sriram V Pemmaraju, and Kasturi R Varadarajan. On the polynomial time computation of equilibria for certain exchange economies.

[48] CoinMarketCap. Today's cryptocurrency prices by market cap. https://coinmarketcap.com/, Dec 2021.

[49] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. *Distributed Computing*, pages 1–17, 2019.

[55] 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.

[56] Ran Duan and Kurt Mehlhorn. A combinatorial polynomial algorithm for the linear Arrow–Debreu market. *Information and Computation*, 243:112–132, 2015.

[57] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[58] 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.

[59] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Yu Xia, Runtian Zhou, and Dahlia Malkhi. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. *arXiv preprint arXiv:2203.06871*, 2022.

[60] 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.

[61] Eyal Hertzog, Guy Benartzi, and Galia Benartzi. Bancor protocol. 2018.

[62] 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.

[63] Kamal Jain, Mohammad Mahdian, and Amin Saberi. Approximating market equilibria. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 98–108. Springer, 2003.

[64] 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.

[65] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

[66] Zoltán Király and Péter Kovács. Efficient implementations of minimum-cost flow algorithms. *arXiv preprint arXiv:1207.6381*, 2012.

[67] 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.

[68] 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.

[69] 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.

[70] James Lovejoy, Cory Fields, Madars Virza, Tyler Frederick, David Urness, Kevin Karwaski, Anders Brownworth, and Neha Narula. A high performance payment processing system designed for central bank digital currencies.

[71] Andrew Makhorin. Glpk (gnu linear programming kit). *http://www.gnu.org/s/glpk/glpk.html*, 2008.

[72] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. http://bitcoin.org/bitcoin.pdf.

[73] 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.

[74] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. *arXiv preprint arXiv:1909.07220*, 2019.

[75] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[76] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[77] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? *arXiv preprint arXiv:2101.05511*, 2021.

[78] Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376*, 2019.

[79] 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.

[80] 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.

[81] Tether. Tether: Fiat currencies on the bitcoin blockchain. https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf. Accessed 12/14/2021.

[82] 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.

[83] 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.

[84] Will Warren and Amir Bandeali. 0x: An open protocol for decentralized exchange on the ethereum blockchain. 2017.

[85] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. *White Paper*, 21, 2016.

[86] 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.

[87] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. *Whitepaper*, 2018.

[88] Yinyu Ye. A path to the Arrow–Debreu competitive market equilibrium. *Mathematical Programming*, 111(1-2):315–348, 2008.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

Appendix A Mathematical Model Underlying SPEEDEX

Mathematically, SPEEDEX's batch computation works via a correspondence between a batch of trade offers and an instance of an "Arrow-Debreu Exchange Market" [28]. SPEEDEX's batch computation is equivalent to the problem of computing equilibria in these markets.

A.1 Arrow-Debreu Exchange Markets

The Arrow-Debreu Exchange Market is a classic concept 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" (the "auctioneer"), not directly with each other.

Definition 1 (Arrow-Debreu Exchange Market)**.** *An Arrow-Debreu Exchange Market consists of a set of goods $[N]$ and a set of agents $[M]$. Every agent j has a utility function $u_j(\cdot)$ and an endowment $e_j \in \mathbb{R}^N_{\geq 0}$.*

When the market trades at prices $p \in \mathbb{R}^N_{\geq 0}$, 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 $x_j \in \mathbb{R}^N_{\geq 0}$ - that is, $x_j = \arg\max_{x:\Sigma x_i p_i \leq s_j} u_j(x)$.

There are countless variants on this definition. Typically the utility functions are assumed to be quasi-convex. Readers familiar with the literature may have seen market model variants which include stock dividents, corporations, production of new goods from existing goods, and multiple trading rounds. SPEEDEX only needs the setup outlined above, with none of these features (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 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 i, $\Sigma_j e_{ij} \geq \Sigma_j x_{ij}$, and x_j is an optimal bundle for agent j.*

Note the subtlety that an equilibrium includes both a set of market prices and a choice of a utility-maximizing set of goods for each agent. If, for example, there are two goods A and B and one unit of each 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 A or

B, and vice versa for the second. It would not be a market equilibrium for both of these agents to purchase a unit of A and no units of B.

A.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 orders of the following form.

Definition 3 (Limit Sell Offer)**.** *A Sell Offer (S, B, e, α) is request to sell e units of good S in exchange for some number k units of good B, subject to the condition that $k \geq \alpha e$.*

The user who submits this offer implicitly says that they value k units of B more than e units of S if and only if $k \geq \alpha e$. Thus, the user's preferences are representable as a simple utility function.

Theorem 2. *Suppose a user submits a sell offer (S, B, e, α). The optimal behavior of this offer (and the user's implicit preferences) is equivalent to maximizing the function $u(x_S, x_B) = \alpha x_B + x_S$ (for x_S, x_B amounts of goods S and B).*

Proof. Such an offer makes no trades if $p_S/p_B < \alpha$ and trades in full if $p_S/p_B > \alpha$.

The user starts with k units of S. In the exchange market model, the user can trade these k units of S in exchange for any quantities x_S of S and x_B of B, subject to the constraint that $p_S x_S + p_B x_B \leq k p_S$.

The function $u(x_S, x_B) = \alpha x_B + x_S$ is maximized, subject to the above constraint, by $(x_B, x_S) = (0, k)$ precisely when $p_S/p_B < \alpha$ and by $(x_B, x_S) = (k p_S/p_B, 0)$ otherwise (and by any convex combination of the two when $p_S/p_B = \alpha$). These allocations correspond exactly to the optimal behavior of a limit sell offer. □

A.3 Existence of a Unique* Equilibrium Prices

Theorem 3. *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 (Theorem 2), and have a nonzero marginal utility on the good being sold.

This means our market instances trivially satisfy condition (*) of Devanur et. al [51]. Existence of an equilibrium with nonzero prices follows therefore from Theorem 1 of [51]. □

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 4. *Suppose there are two equilibria (p, x) and $(p', x'$ and there exist two assets A and B for which $p_A/p_B \neq p'_A/p'_B$.*

Then it must be the case that there is a partitioning of the assets $\mathcal{A}, \mathcal{B}$ with $A \in \mathcal{A}, B \in \mathcal{B}$ such that both equilibria include no trading activity across the partition.

Proof. Consider the set of offers trading from one asset a to another asset b. Observe that the amount, say $z_{ab}(p)$, of a that is available for sale for b decreases as the exchange rate p_a/p_b decreases. Note further that the quantity $(p_a/p_b) z_{ab}(p)$ decreases as p_a/p_b decreases, and that this decrease is strict unless $z_{ab}(p)$ is 0. At equilibrium, $x_{ab} = z_{ab}(p)$.

A technicality: Because some offers might be indifferent to trading at an exchange rate, $z_{ab}(\cdot)$ is in fact a set-valued function, with output $[z_l(p_a/p_b), z_h(p_a/p_b)]$ for $z_l(\cdot)$ the lower bound on the amount of a that must be sold, and $z_h(\cdot)$ the upper bound. Observe that $z_l(r_1) \geq z_h(r_2)$ when $r_1 > r_2$. At equilibrium, $x_{ab} \in z_{ab}(p)$.

Suppose that there exists a pair of assets (A, B) with $p_A/p_B \neq p'_A/p'_B$ (wlog $p_A/p_B < p'_A/p'_B$). Then there must be a set of assets C for which every asset pair (c, d) with $c \in C$, $d \notin C$ with $p_c/p_d < p'_c/p'_d$, and which has $A \in C$, $B \notin C$.

Let F be the set of these edges.

At equilibrium, we must have that $\Sigma_{(c,d) \in F} p_c x_{cd} - p_d x_{dc} = 0$. But for all of these edges (c, d) (again, $c \in C$), we must have that

$$p_c/p_d x_{cd} \leq p'_c/p'_d x'_{cd}$$

and

$$x_{dc} \geq x'_{dc}$$

Combining these gives

$$p_c x_{cd} - p_d x_{dc} \leq (p'_c x'_{cd} - p'_d x'_{dc}) p_d/p'_d$$

Note that we can always rescale the prices of an equilibrium to find a new equilibrium with the same allocation, and so it is without loss of generality to assume $p'_d > p_d$ for all assets d.

Hence,

$$p_c x_{cd} - p_d x_{dc} \leq (p'_c x'_{cd} - p'_d x'_{dc})$$

and thus

$$\begin{aligned} 0 &= \Sigma_{(c,d) \in F} p_c x_{cd} - p_d x_{dc} \\ &\leq \Sigma_{(c,d) \in F} p'_c x'_{cd} - p'_d x'_{dc} \end{aligned}$$

The inequality must be strict if there is any $(c, d) \in F$ with $x_{cd} > 0$ (note that if there is some edge $(c, d) \in F$ with $x_{dc} > 0$, then there must also be an edge $(c', d') \in F$ with $x_{c'd'} > 0$ also). Hence, there can be no trading activity across edges in F.

Hence, (p', x') can only be an equilibrium if there exists a partitioning of the assets that separates A and B, and for which there is no trading activity between the sets in either equilibrium.

□

Corollary 1. *Let (p, x) be an equilibrium.*

Construct an undirected graph $G = (V, E)$ with one vertex for each asset, and an edge $e = (A, B) \in E$ if, at equilibrium, any A is sold for B or any B is sold for A (that is, if $x_{AB} > 0$).

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 (A, B), $p_A/p_B = p'_A/p'_B$. By Theorem 4, if this did not hold, then there would exist a set of edges (a, b) that partitions the vertex set V for which $x_{ab} = 0$. This would contradict the assumption that G is connected.

□

Appendix B 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_A on each asset A, and a trade amount x_{AB} denoting the amount of A sold in exchange for B.

Formally, we say that the result of a batch price computation is (ε, μ)-approximate if:

1. Asset conservation is preserved with a ε commission. The amount of A sold to the auctioneer, $\Sigma_B x_{AB}$, must exceed the amount of A bought from the auctioneer, $\Sigma_B (1-\varepsilon) \frac{p_B}{p_A} x_{BA}$.
2. No offer trades outside of its limit price. That is to say, an offer selling A for B with a limit price of r cannot execute if $\frac{p_A}{p_B} < r$.
3. No offer with a limit price "far" from the batch exchange rate does not trade. That is to say, an offer selling A for B with a limit price of r must trade in full if $r < (1-\mu)\frac{p_A}{p_B}$. 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. [46], 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).

Appendix C Tâtonnement Modifications

C.1 Price Update Rule

One significant algorithmic difference between the Tâtonnement implemented within SPEEDEX and the Tâtonnement described in Codenotti et al. [46] 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:

$$p_i \leftarrow p_i + Z_i(p)\delta \tag{1}$$

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.

$$p_i \leftarrow p_i(1 + Z_i(p)\delta) \tag{2}$$

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.

$$p_i \leftarrow p_i(1 + p_i Z_i(p)\delta) \tag{3}$$

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, $\Sigma_i (p_i Z_i(p))^2$; 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 [26, 35] with a weakened termination condition.

$$p_i \leftarrow p_i(1 + p_i Z_i(p)\delta_t) \tag{4}$$

Finally, we normalize adjustments by a trade volume factor ν_i. 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 [33]).

ν_i 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:

$$p_i \leftarrow p_i(1 + p_i Z_i(p)\delta_t \nu_i) \tag{5}$$

The step size is represented internally as a 64-bit integer and a constant scaling factor. As mentioned in §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.

C.1.1 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. [35]). Devanur et. al [51] even give a convex objective function for computing exchange market equilibria, which we reproduce below (in a simplified form):

$$\Sigma_{i:mp_i < \frac{p_{S_i}}{p_{B_i}}} p_{S_i} E_i \ln(mp_i \frac{p_{S_i}}{p_{B_i}}) - y_i \ln(mp_i) \tag{6}$$

for mp_i the minimum limit price of an offer i that sells E_i units of good S_i and buys good B_i, and $y_i = x_i p_{S_i}$ for x_i the amount of S_i sold by the offer to the market.

This objective is accompanied by an asset conservation constraint for each asset A:

$$\Sigma_{i:S_i=A} y_i = \Sigma_{i:B_i=A} y_i \tag{7}$$

However, unlike the problem formulation in [51], 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 $y_i = p_{S_i} E_i$ if $mp_i < \frac{p_{S_i}}{p_{B_i}}$ and 0 otherwise (although §C.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

$$\Sigma_A \lambda_A (\Sigma_{i:S_i=A} y_i(p) - \Sigma_{i:B_i=A} y_i(p)) \tag{8}$$

for a set of langrange multipliers $\{\lambda_A\}$.

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 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 §C.2).

However, observe that equation 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 8 with respect to λ_A is the amount by which (the additive version of) Tâtonnement updates p_A. This might suggest using p_A in place of λ_A in equation 8. However, that search heuristic performs extremely poorly.

C.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 §B 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 for every offer (selling A for B) with a limit price below $(1-\mu)\frac{p_A}{p_B}$ trades in full, and those with limit prices above $\frac{p_A}{p_B}$ trade not at all.

As such, SPEEDEX has the flexibility to specify offer behavior on the gap between $(1-\mu)\frac{p_A}{p_B}$ and $\frac{p_A}{p_B}$. Instead of a step function, SPEEDEXlinearly interpolates across the gap. That is to say, if $\alpha = \frac{p_A}{p_B}$, we say that an offer with limit price $(1-\mu)\alpha \leq \beta \leq \alpha$ sells a $\frac{\alpha-\beta}{\mu\alpha}$ 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 2, particularly why the price computation problem gets increasingly difficult as μ decreases.

C.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.

Appendix D 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 A (as output from Tâtonnement) as p_A, and the amount of A sold in exchange for B as x_{AB}. We will also denote the two forms of approximation error as ε and μ, as defined in §B.

To maintain asset conservation, the linear program must

satisfy the following constraint for every asset A:

$$\Sigma_B x_{AB} \geq \Sigma_B (1-\varepsilon)\frac{p_B}{p_A} x_{BA}$$

Define U_{AB} to be the upper bound on the amount of A that is available for sale by all offers with in the money limit prices (i.e. limit prices below $\frac{p_A}{p_B}$), and define L_{AB} to be the lower bound on the amount of A that must be exchanged for B if SPEEDEX is to be μ-approximate (i.e. execute all offers with minimum prices below $(1-\mu)\frac{p_A}{p_B}$, as described in §B).

Then the linear program must also satisfy the constraint, for every asset pair (A,B),

$$L_{AB} \leq x_{AB} \leq U_{AB}$$

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:

$$\Sigma_{A,B} p_A x_{AB}$$

Putting this all together gives the following linear program:

$$\max\ \Sigma_{A,B} p_A x_{AB} \tag{9}$$

$$s.t.\ p_A L_{AB} \leq p_A x_{AB} \leq p_A U_{AB}(p) \quad \forall (A,B),\ (A \neq B) \tag{10}$$

$$p_A \Sigma_{B\in[N]} x_{AB} \geq (1-\varepsilon)\Sigma_{B\in[N]} p_B x_{BA} \quad \forall A \tag{11}$$

From the point of view of the linear program, p_A is a constant (for each asset 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 §6). In these cases, we set the lower bound on each x_{AB} to be 0 instead of L_{AB}. This change makes the program always feasible (i.e. an assigment of each variable to 0 satisfies the constraints).

Observe that as written, every instance of the variable x_{AB} appears adjacent to p_A. We can simplify the program by replacing each occurrence of $p_A x_{AB}$ by a new variable y_{AB}. After solving the program, we can compute x_{AB} as $\frac{y_{AB}}{p_A}$.

This substitution gives the following linear program:

$$\max\ \Sigma_{A,B} y_{AB} \tag{12}$$

$$s.t.\ p_A L_{AB} \leq y_{AB} \leq p_A U_{AB}(p) \quad \forall (A,B),\ (A \neq B) \tag{13}$$

$$\Sigma_{B\in[N]} y_{AB} \geq (1-\varepsilon)\Sigma_{B\in[N]} y_{BA} \quad \forall A \tag{14}$$

The Stellar Development Foundation plans to charge no transaction commission (i.e. set ε to 0) in its SPEEDEX deployment. This makes the linear program into an instance of the maximum circulation problem (i.e. variable y_{AB} denotes the "flow" from vertex A to vertex B) and makes the constraint matrix "totally unimodular", which means it has an integral solution and can be solved exactly by specialized algorithms (such as those outlined in [66]). Some of these algorithms run substantially faster than more general simplex-based solvers.

Appendix E Market Structure Decomposition

Suppose that the set of goods could be partitioned between a set of "pricing assets", 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 5. *Let A be the set of pricing assets and B the set of stocks. A stock $s \in B$ is traded with asset $a(s) \in A$.*

Suppose (p,x) is an equilibrium for the restricted market instance considering only the pricing assets. For each $s \in B$, let (r,y) be an equilibrium for the restricted market instance considering only s and $a(s)$.

Then (p',x') is an equilibrium for the entire market instance, where

1. $p'_a = p_a$ *for* $a \in A$
2. $p'_s = (r_s/r_{a(s)})p_{a(s)}$
3. $x'_{ab} = x_{ab}$ *for* $a,b \in A$
4. $x'_{s,a(s)} = y_{s,a(s)}$
5. $x' = 0$ *otherwise*

Proof. More generally, let G be a graph whose vertices are the traded assets and which contains an edge (a,b) if a and 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).

□

Appendix F Alternative Batch Solving Strategies

F.1 Convex Optimization

We implemented the convex program of Devanur et. al [51] directly, using the CVXPY toolkit [53] backed by the ECOS convex solver [55]. The exact 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, 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 (a few seconds and a few tenths of a second, respectively).

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 §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 non-differentiable. The demand smoothing tactic of §C.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 [51] while maintaining sufficient smoothness and numerical stability is an interesting open problem.

F.2 Mixed Integer Programming

Gnosis (Walther, [5]) 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 [5] 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.

Appendix G 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 §8.2, 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 §C.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)

$$\Sigma_{i:mp_i \leq p} E_i \tag{15}$$

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 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)

$$\Sigma_{i:mp_i < p(1-\mu)} E_i + \Sigma_{i:p(1-\mu) \leq mp_i \leq p} E_i * (p - mp_i)/(p\mu) \tag{16}$$

We can rearrange the second term of the summation into

$$1/(p\mu)\Sigma_{i:p(1-\mu) \leq mp_i \leq p}(pE_i - E_i mp_i) \tag{17}$$

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 15 and

$$\Sigma_{i:mp_i < p} mp_i E_i \tag{18}$$

The division in equation 16 can be avoided by recognizing that Tâtonnement normalizes all asset amounts by asset valuations (so equation 16 is always multiplied by p).

Appendix H 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".

Example 2 (Limit Buy Offer)**.** *A user offers to buy 100 USD in exchange for EUR, selling as few EUR as possible and only if one EUR trades for at least* 1.1 *USD.*

These offers unfortunately do not satsify a property known as "Weak Gross Substitutability" (WGS, see e.g. [46]). 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 satisfy this property, but limit buy offers do not.

Example 3. *The demand of the offer in of example 2, when* $p_{EUR} = 2$ *and* $p_{USD} = 1$*, is* $(-50EUR, 100USD)$.

If p_{USD} *rises to* 1.6*, then the demand for the offer is* $(-80EUR, 100USD)$.

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. [41] 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 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.

Appendix I Additional Implementation Details

I.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 (A, B), we build a trie storing offers selling asset A in exchange for 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 replicas (to validate consensus, and check for errors), and construct short proofs for users about exchange state.

I.2 Data Storage and Persistence

Processing transactions in a nondeterministic order complicates recovery from a database snapshot where a block has been partially applied. Therefore, we would like a system that can update all state modified in one block in one atomic transaction. We also want a database where elements can be accessed directly in memory, and that supports concurrent modification.

SPEEDEX uses a combination of an in-memory cache and an ACID-compliant database (LMDB [43]). This choice suffices for our experiments, but a database that persists data in epochs, like Silo [82], might improve performance.