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

Geoffrey Ramseyer, Ashish Goel, and David Mazières, *Stanford University*

https://www.usenix.org/conference/nsdi23/presentation/ramseyer

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

SPEEDEX:
A Scalable, Parallelizable, and Economically Efficient Decentralized 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 $\mathcal{A}$ to asset $\mathcal{B}$ always receives as good a price as trading through some third asset such as USD. Finally, it prevents certain front-running attacks that would otherwise increase the effective bid-ask spread for small traders. SPEEDEX's key design insight is its use of an Arrow-Debreu exchange market structure that fixes the valuation of assets for all trades in a given block of transactions. We construct an algorithm, which is both asymptotically efficient and empirically practical, that computes these valuations while exactly preserving 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 prototyped but not yet merged within the Stellar blockchain, one of the largest Layer-1 blockchains.

1 Introduction

Digital currencies are moving closer to mainstream adoption. Examples include central bank digital currencies (CBDCs) such as China's DC/EP [90], commercial efforts [65, 75], and many decentralized-blockchain-based stablecoins such as Tether [104], Dai [9], and USDC [17]. These currencies vary wildly in terms of privacy, openness, smart contract support, performance, regulatory risk, solvency guarantees, compliance features, retail vs. wholesale suitability, and centralization of the underlying ledger. Because of these differences, and because financial stability demands different monetary policy in different countries, we cannot hope for a one-size-fits-all global digital currency. Instead, to realize the full potential of digital currencies (and digital assets in general), we need an ecosystem where many digital currencies can efficiently interoperate.

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

The ideal digital currency exchange should, at minimum,

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

Scalability is crucial for a piece of financial infrastructure that must last far into the future, as the number of individuals transacting internationally continues to grow. Furthermore, the above feature list is by no means complete; a deployment may want any number of additional features, such as persistent logging, simplified payment verification [89], or integrations with legacy systems, each of which slows down the system's performance. Scalability, viewed from another angle, enables the system to add features without decreasing overall transaction throughput (at the cost of additional compute hardware).

The gold standard for avoiding centralized control is a *decentralized exchange*, or DEX: a transparent exchange implemented as a deterministic replicated state machine maintained by many different parties. To prevent theft, a DEX requires all transactions to be digitally signed by the relevant asset holders. To prevent cheating, replicas organize history into an append-only blockchain. Replicas agree on blockchain state through a Byzantine-fault tolerant consensus protocol, typically some variant of asynchronous or eventually synchronous Byzantine agreement [46] for private blockchains

or synchronous mining [89] for public ones.

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

Equality of Access In existing exchange designs, users with low-latency connections to an exchange server (centralized or not) can spy on trades incoming from other users and *front-run* these trades. For example, a front-runner might spy an incoming sell offer, and in response, send a trade that buys and immediately resells an asset at a higher price [38,82]. In a blockchain, where a block of trades is either finalized entirely or not at all, this front-running can be made risk-free. More generally, some users form special connections with blockchain operators to gain preferential treatment for their transactions [55]. 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 [95].

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

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

Unfortunately, folk wisdom holds that DEXes cannot scale beyond a few thousand transactions per second. Naïve parallel execution would not be replicable across different blockchain nodes. This wisdom has led to many alternative blockchain scaling techniques, such as off-chain trade matching [108], automated market-makers [24], transaction rollup systems [15, 19], and sharded blockchains [6] or side-chains [92]. These approaches either trust a third party to ensure that orders are matched with the best available price, or sacrifice the ability to set traditional limit orders that only sell at or above a certain price (reducing market liquidity). Offchain rollup systems, sharded chains, and side-chains further fragment market liquidity, leading to cross-shard arbitrage and worse exchange rates for traders.

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

1.1 SPEEDEX: Towards an Ideal DEX

This paper disproves the conventional wisdom about on-chain DEX performance. We present SPEEDEX, a fully on-chain decentralized exchange that meets all of the desiderata outlined above. SPEEDEX gives every user an equal level of access (thereby eliminating a widespread class of risk-free front-running), eliminates internal arbitrage opportunities (thereby making optimal use of liquidity available on the DEX), and is capable of processing over 200,000 transactions per second when deployed on 48-core machines (Figure 3). SPEEDEX is designed to scale further when given more hardware.

Like most blockchains, SPEEDEX processes transactions in blocks—in our case, a block of 500,000 transactions every few seconds. Its fundamental principle is that transactions in a block commute: a block's result is identical regardless of transaction ordering, which enables efficient parallelization [51].

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

This design provides two additional economic advantages. First, the exchange offers liquid trading between every asset pair. Users can directly trade any asset for any other asset, and the market between these assets will be at least as liquid as the most liquid market path through intermediate reserve currencies. Second, SPEEDEX eliminates a class of front-running that is widespread in modern DEXes. No exchange operator or user with a low-latency network connection can buy an asset and resell it at a higher price, within the same block. (Note that this is not every type of front-running; §8 and §10 contrast SPEEDEX's guarantees with those of other mitigations, and how they can be combined.)

Furthermore, this economic design enables a scalable systems design that is not possible using traditional order-matching semantics. Unlike every other DEX, the operation of SPEEDEX is efficiently parallelized, allowing SPEEDEX to scale to transaction rates far beyond those seen today. Transactions within a block commute with each other precisely because trades all happen at the same shared set of exchange rates. This means that the transaction processing engine has no

need for the sequential read-modify-update loop of traditional orderbook matching engines. Account balances are adjusted using only hardware-level atomics, rather than locking.

1.2 SPEEDEX Overview

SPEEDEX is not a blockchain itself; rather, it is a DEX component that can be integrated into any blockchain. A copy of the SPEEDEX module should run inside every replica of a blockchain using the system. SPEEDEX does not depend on any specific property of a consensus protocol, but automatically benefits from throughput advances in consensus and transaction dissemination (such as [56]). SPEEDEX heavily uses concurrency and benefits from uninterrupted access to CPU caches, and as such is best implemented directly within blockchain node software (instead of as a smart contract).

We implemented SPEEDEX within a custom blockchain using the HotStuff consensus protocol [115]; this implementation provides the measurements in this paper. We created a second implementation as a component of the Stellar blockchain [84], which is considering a Layer-1 SPEEDEX deployment.

Implementing SPEEDEX introduces both theoretical algorithmic challenges and systems design challenges. The core algorithmic challenge is the computation of the batch prices. This problem maps to a well-studied problem in the theoretical literature (equilibrium computation of *Arrow-Debreu Exchange Markets*, §A.1); 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 [53]) that, in practice, efficiently approximates batch clearing prices. We then explicitly correct approximation error with a follow-up linear program.

Our algorithm's runtime is largely independent of the number of limit orders—each Tâtonnement query has a runtime of $O(\#\text{assets}^2 \cdot \lg(\#\text{offers}))$ and the linear program has size $O(\#\text{assets}^2)$. This gives a crucial algorithmic speedup because in the real world, the number of currencies is much smaller than the number of market participants. (The experiments of §6 and §7 use 50 assets and tens of millions of open offers.)

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

In recent years, the economics literature has begun discussing the use of batched trading systems in traditional markets to combat front-running and externalities associated with high-frequency trading [30, 41, 43]. This literature focuses only on the case of trading between two assets (where price computation is simple) or where all trades use a single *numeraire* currency [44]. Our contribution to this line of work is to demonstrate the feasibility of a batch trading system that exchanges many assets and many numeraire currencies simultaneously, thereby expanding the design space of implementable market structures.

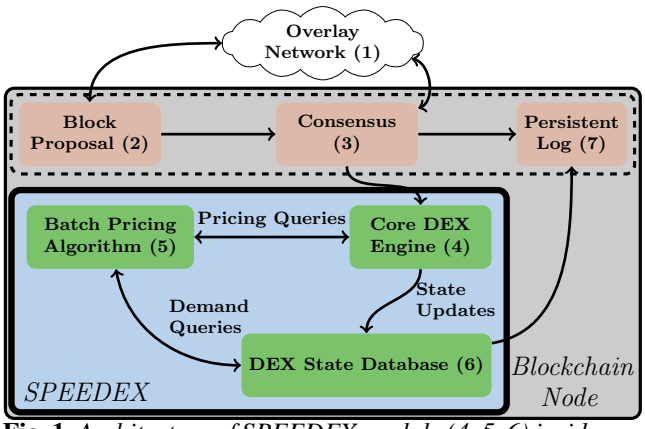

Fig. 1. *Architecture of SPEEDEX module (4, 5, 6) inside one blockchain node.*

2 System Architecture

SPEEDEX is an asset exchange implemented as a replicated state machine in a blockchain architecture (Fig. 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 (on the order of seconds), such as BA⋆ [71], SCP [84], or HotStuff [115].

The implementation evaluated here uses HotStuff [115], while the the Stellar blockchain implementation relies on Stellar's existing consensus protocol, SCP [88].

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

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

What makes SPEEDEX different from existing DEXes is the manner in which it processes new orders. Traditional

exchanges process trades sequentially, implicitly computing a matching between limit orders. SPEEDEX, by contrast, processes trades in batches (typically, one batch would consist of all of the limit orders in one block of the blockchain).

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

Logically, when the SPEEDEX core engine (Fig. 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: one offer to buy an asset is matched with the lowest priced seller, and the next offer to buy is matched against the second-lowest priced seller, and so on. Hence, every trade can occur at a slightly different exchange rate.

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

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

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

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

[1] And technical conditions (§A.3), e.g. everything clears an empty market.

Theorem 1 is a restatement of a general theorem of Arrow-Debreu exchange market theory [57] (§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).

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

Unfortunately, we are not aware of a practical method to compute clearing prices exactly. (The number of bits required to represent exact clearing prices may be extremely large [57], and in a natural extension of the SPEEDEX model [96] the clearing prices are not even rational.) SPEEDEX therefore uses *approximate* clearing prices.

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

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

2.2 Design Properties

Computational Scalability SPEEDEX's commutative semantics allow effective parallelization of DEX operation. Because transactions within a block are not semantically ordered, DEX state is identical regardless of the order in which transactions are applied. This exact replicability is, of course, required for a *replicated* state machine. The order-independence also means SPEEDEX transactions can be executed in parallel by all available CPU cores despite the fact that thread interleaving is nondeterministic in multicore machines. Almost all coordination occurs via hardware-level

atomics (e.g., atomic add on 64-bit integers) without spinlocks.

SPEEDEX stores balances in accounts, rather than in discrete, unspent coins (often called "UTXOs"). It also supports single-currency payment operations, which are simpler than DEX trading. Hence, SPEEDEX disproves the popular belief [85, 97] that account-based ledgers are not compatible with horizontal scalability.

No risk-free front running Well-placed agents in real-world financial markets can spy on submitted offers, notice a new transaction T, and then submit a transaction T' (that executes before T) that buys an asset and re-sells it to T at a slightly higher price. In some blockchain settings, T' can be done as a single atomic action [55]. 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) [55] phenomenon, in which block producers reorder transactions within a block for their own profit (or in exchange for kickbacks). By eliminating the ordering of transactions within a block, SPEEDEX eliminates a large source of MEV. However, this does not eliminate every type of front-running manipulation, such as delaying victim transactions to a future block (see §8).

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

3 Commutative DEX Semantics

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

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

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

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

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

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

SPEEDEX must also ensure that no account is overdrafted. That is to say, after processing all transactions in a block, the unlocked balance of every account must be nonnegative (where an open offer locks the offered amount of an asset for the duration of its lifetime). Unlike most distributed ledgers, SPEEDEX cannot simply deem the second of two conflicting transactions to fail—after all, transactions have no ordering. Instead, our implementation requires a block proposer to ensure that a block cannot cause overdrafts; every node rejects blocks that violate this property. To generate valid blocks, proposers use a conservative process outlined in §K.6. The design requires passing information from the SPEEDEX database (Fig 1, 6) to the proposal module (Fig 1, 2).

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

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 algorithm is based on the iterative Tâtonnement process from this literature [53].

However, the runtimes of the theoretical algorithms scale very poorly, both asymptotically and empirically. They also output approximate equilibria for notions of approximation that violate the two fundamental constraints above (for example, Definition 1 of [53] permits equilibria to mint new assets and to steal from a user).

We develop a novel algorithm for computing equilibria that runs efficiently in practice (§6) and explicitly ensures that (1) asset amounts are conserved and (2) every offer trades at exactly the market prices, and only if the offer's limit price is at or below the batch exchange rate. First, Tâtonnement approximates clearing prices (§5). We show that the structure of the types of trades in SPEEDEX lets each iteration run in time logarithmic in the number of open limit offers (via a series of binary searches), giving an algorithm asymptotically faster than that within the theoretical literature.

We then explicitly correct for the approximation error with a linear program (§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, our algorithm outputs the following:

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

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

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

5 Price Computation: Tâtonnement

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

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

Example 1. *Suppose that a limit order offers to sell 100 USD for EUR with a minimum price of* 0.8 *EUR per USD. If the candidate prices are such that* $\alpha = \frac{p_{USD}}{p_{EUR}} > 0.8$*, then the limit order would like to trade, and its demand is* $(-100\ USD, 100\alpha\ EUR)$*. Otherwise, its demand is* $(0\ USD, 0\ EUR)$*.*

Iterative Price Adjustment. If more units of an asset are demanded from the auctioneer than are supplied to it (a positive net demand, meaning a deficit for the auctioneer), then the auctioneer raises the price of the asset. Otherwise, the auctioneer has a surplus, so it lowers the price of the asset. Implementing this process effectively requires careful numerical normalization in response to differences in prices and trade volumes, which we describe in detail in §C.1.

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

1. The auctioneer has no deficits (assets are conserved)
2. No offer executes outside its limit price bound
3. Every offer with a limit price more than a $(1-\mu)$ factor below the auctioneer's exchange rate executes in full.

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

5.1 Efficient Demand Queries

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

This naïve loop appears to be required for the (more general) problem instances studied in the theoretical literature. However, all of the offers in SPEEDEX are traditional limit orders that sell one asset in exchange for one other asset at some limit price. An offer with a lower limit price always trades if an offer with a higher limit price trades. Therefore, SPEEDEX groups offers by asset pair and sorts offers by their limit prices. We drive the marginal cost of this sorting to near zero by using an offer's limit price as the leading bits (in big-endian) of the keys in our Merkle tries (§K.5).

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

5.2 Multiple Tâtonnement Instances

§C describes several other Tâtonnement adjustments that help it respond well to a wide variety of market conditions. Some of these adjustments are parametrized (such as how quickly one should adjust the candidate prices); rather than pick one set of control parameters, we run several instances of Tâtonnement in parallel and take whichever finishes first as the result. (In the case of a timeout, we choose the set of prices that minimizes the *unrealized utility* [§6.2].) SPEEDEX includes the output of Tâtonnement and the subsequent linear program in the headers of proposed blocks (§K.3).

6 Evaluation: Price Computation

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

6.1 Accuracy and Orderbook Size

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

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

Fig. 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). To put these fee rates in context, BinanceDex [1] charged a fee of either $0.1\% \approx 2^{-10}$ or $0.04\% \approx 2^{-11.3}$. Uniswap [24, 25] charges 1%, 0.3%, or 0.05% ($\sim 2^{-6.6}$, $\sim 2^{-8.4}$, and $\sim 2^{-11}$, respectively), and Coinbase charges 0.5% to 4% [4] ($\sim 2^{-7.6}$ to $\sim 2^{-4.6}$).

Though our experiments rarely experienced Tâtonnement timeouts, Tâtonnement timeouts caused by sparse orderbooks may be self-correcting: If SPEEDEX proposes suboptimal prices, fewer offers will find a counterparty and trade. When fewer offers clear in one block, more are left to facilitate

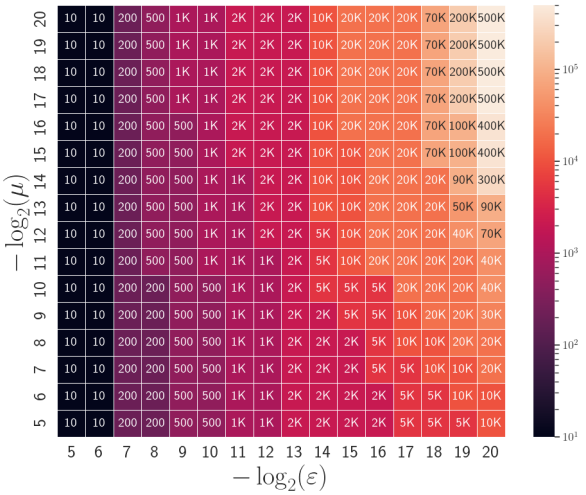

Fig. 2. *Minimum number of offers needed for Tâtonnement to run in under* 0.25 *seconds (Smaller is better. Times averaged over 5 runs). The x axis denotes offer behavior approximation quality (μ), and the y axis denotes the commission (ε).*

Tâtonnement in the next block. §F describes an alternative algorithm that is effective on small batches.

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 $\mathcal{A}$ (and buys asset $\mathcal{B}$) with probability proportional to the relative volume of asset $\mathcal{A}$ (and asset $\mathcal{B}$, conditioned on $\mathcal{A} \neq \mathcal{B}$) on day i, and demands a minimum price close to the real-world exchange rate on day i. The extreme volatility of cryptocurrency markets and variation between these 50 assets make this dataset particularly difficult for Tâtonnement. To further challenge Tâtonnement, we use a smaller block size of $\sim$30,000 (compared to 500,000 in §7).

The experiment charged a commission of $\varepsilon = 2^{-15} \approx$

0.003%, and attempted to clear offers with limit prices more than $1-\mu$ below the market prices, for $\mu = 2^{-10} \approx 0.1\%$ (§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, every Tâtonnement iteration continues to refine the price of every asset. This is why Tâtonnement actually gives more accurate results in the batches it found challenging. A deployment might enforce a minimum number of Tâtonnement rounds.

Qualitatively, Tâtonnement correctly prices assets with high trading volume and struggles on sparsely traded assets (as might be expected from Fig. 2). Tâtonnement also adjusts its price adjustment rule in response to recent market conditions (§C.1), a tactic which is less effective on volatile assets.

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

7 Evaluation: Scalability

We ran SPEEDEX on four r6id.24xlarge instances in an Amazon Web Services datacenter. Each instance has 48 physical CPU cores divided over two Intel Xeon Platinum 8375CL chips (32 total cores per socket, 24 of which are allocated to our instances), running at 2.90Ghz with hyperthreading enabled, 768GB of memory, 4 1425GB NVMe drives connected in a RAID0 configuration. We use the XFS filesystem [102]. These experiments use the HotStuff consensus protocol [115], and do not include Byzantine replicas or a rotating leader.

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

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

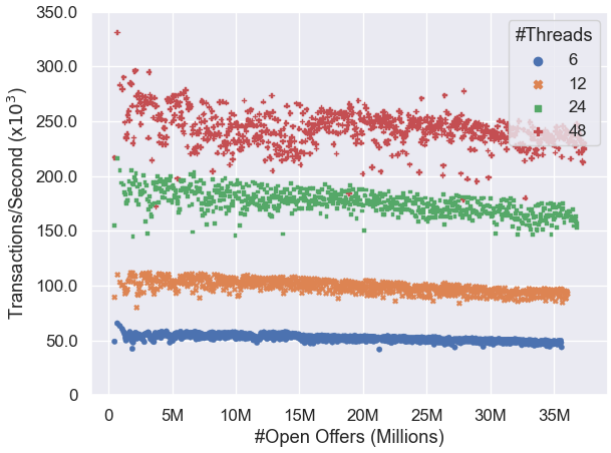

Fig. 3. *Transactions per second on SPEEDEX, plotted over the number of open offers.*

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

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

Every five blocks, the exchange commits its state to persistent storage in the background (via LMDB [50], §K.2).

Performance Measurements. Fig. 3 plots the end-to-end transaction throughput rate of SPEEDEX as the number of worker threads inside SPEEDEX increases. The x-axis plots the number of open offers on the exchange.

Most importantly, Fig. 3 demonstrates that SPEEDEX can efficiently use its available CPU hardware. The speedup is near-linear, until the number of threads approaches the number of CPU cores—from 6 to 12, $\sim 1.9x$, from 12 to 24, $\sim 1.8x$, and from 24 to 48, $\sim 1.4x$. The thread counts are only for the number of threads directly for SPEEDEX's critical path, and not for many of the tasks that the implementation must perform in the background, such as logging data to persistent storage (logging the account database uses 16 threads), consensus, and garbage collection, and these threads begin to contend with SPEEDEX as the number of SPEEDEX worker threads increases.

Secondly, Fig. 3 demonstrates the scalability of SPEEDEX with respect to the number of open offers. The number of open

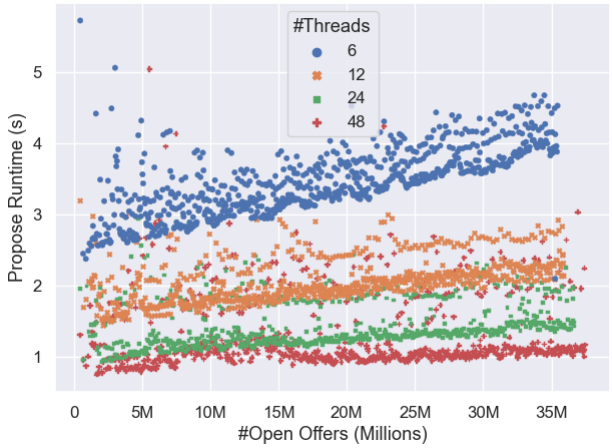

Fig. 4. *Time to propose and execute a block, plotted over the number of open offers.*

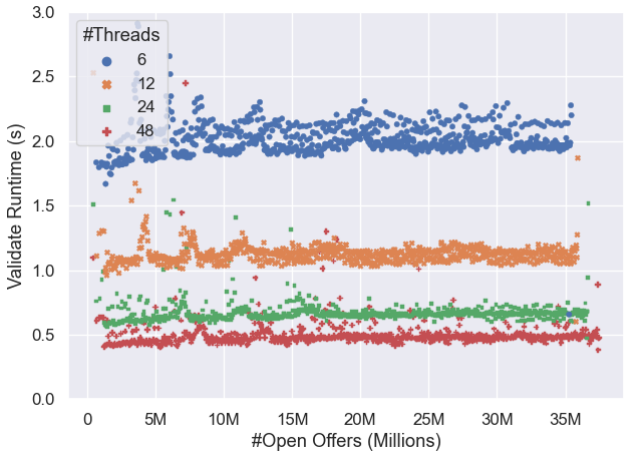

Fig. 5. *Time to validate and execute a proposal, plotted over the number of open offers (measurements from one replica).*

offers SPEEDEX works with in these experiments is already quite large, but most importantly, as the number of open offers goes from 0 to the 10s of millions, SPEEDEX's transaction throughput falls by only $\sim 10\%$. This slowdown is primarily derived from a Tâtonnement optimization (the precomputation outlined in §9.2). Tâtonnement is the one part of SPEEDEX that cannot be arbitrarily parallelized, so we design our implementation towards making it as fast as possible. An implementation might skip this work in some parameter regimes.

To focus on the performance of SPEEDEX, Figs. 4 and 5 plot the time to propose and execute blocks, and to validate and execute proposals, respectively, when we disable signature verification (which is trivial to parallelize). First, note that both proposal and validation scale with the number of threads; validation scales better than proposal due to the aforementioned Tâtonnement optimization. Second, note that validating and executing a proposal from another replica is substantially faster than proposing a block; this lets a replica that is somehow delayed catch up.

The runtime variation in Fig. 4 results from the fact that SPEEDEX without signature verification runs too quickly for our persistent logging implementation.

SPEEDEX is not a consensus protocol, and these experiments (one consensus invocation every few seconds) do not come close to stressing the consensus throughput of Hotstuff. However, network bandwidth requirements necessarily scale (at least) linearly with transaction rate. Recent work, such as [56, 79, 113], develops consensus protocols that maximally use available network bandwidth. However, integrating SPEEDEX with any consensus protocol requires understanding the tradeoffs between batch size, transaction rate, and consensus frequency. Fig. 6 plots this tradeoff running SPEEDEX on the same transaction workload as in Fig. 3. We also ran SPEEDEX with more replicas on different hardware and observed the same scalability trends, as outlined in §L (albeit with lower overall throughput on weaker hardware).

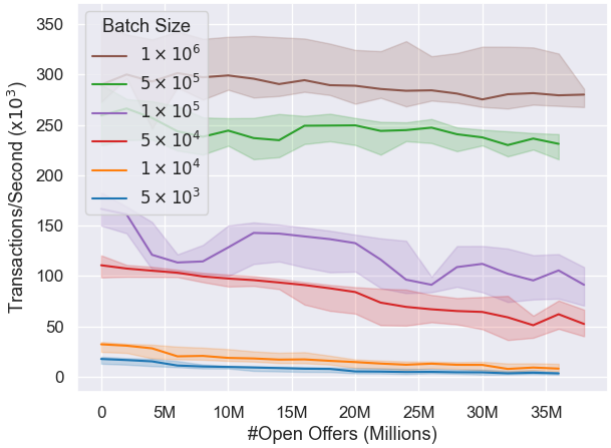

Fig. 6. *Median transaction rates, varying block size and number of open offers (grouped into buckets of 2M). Shaded areas plot 10th to 90th percentiles.*

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

7.1 Alternative Scaling Techniques

Traditional Exchange Semantics. The core logic of just an exchange system can be implemented extremely efficiently with almost no code. The logic of the constant product market maker UniswapV2 [24], for example, is less than 10 lines of simple arithmetic code. An orderbook-based exchange requires more code but can still be made very fast, as most operations modify only a small number of data objects. We implemented a bare-bones orderbook exchange with two assets using the same data structures as in SPEEDEX—each transaction checks the orderbook for a matching offer or offers and either makes appropriate transfers or adds the new offer to the

orderbook. These operations are extremely fast when the number of accounts is small; our implementation runs ∼1.7 million of these transactions per second when there are only 100 accounts. However, every database lookup becomes slower as the as the number of accounts grows; when there are 10 million accounts in the database (as in the above experiments), throughput falls 8x to ∼210,000 per second. Yet that is before adding all of the other SPEEDEX features one needs in a real DEX, such as state hashes, transaction fees, structures for simple payment verification [89], replication, or durable logging. The scalability of the full SPEEDEX implementation lets it surpass that rate even when slowed down by all of these features.

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

Optimistic Concurrency Control. A widely explored class of alternative designs for parallel transaction execution use optimistic concurrency control, and of these approaches the most closely related state of the art design appears to be Block-STM [70], which is deployed in Aptos [22]. This approach optimistically executes batches of transactions, retrying after conflicts as necessary.

We therefore design the measurements of Fig. 7 to mirror the experiments in [70]. The "Aptos p2p" transactions in [70] are payments between two random accounts, and consist of 8 reads of 5 writes. Each of our payments consists of two data reads (source account public key and last committed sequence number), two atomic compare_exchange operations (subtract payment and fee from source), an atomic fetch_xor (reserve sequence number), and an atomic fetch_add (add payment to destination)—implemented without atomics, this would be 6 reads and 4 writes. All payments are of the same asset.

Fig. 7 plots the throughput rates of SPEEDEX on this transaction workload for the parameter settings measured in Block-STM (Figs. 7 and 8, [70]). Note that for large batch sizes, the transaction throughput is largely independent of the number of accounts, even though every transaction in the two account setting contends with every other transaction. Furthermore, unlike Block-STM, SPEEDEX achieves near-linear scalability on sufficiently large batches. For small batch sizes, a large number of accounts actually slows down SPEEDEX, largely due to increased sensitivity to cache performance and our system's NUMA (two socket) architecture on small timescales. We also ran this experiment on a single-socket system (an AWS c5a.16xlarge, as in [70]), and found only negligle impact of the number of accounts on throughput. Fig. 7 was run with hyperthreading disabled, to compare against Block-STM experiments. The rest of our experiments were run with hyperthreading enabled (because of the many background tasks in SPEEDEX); enabling hyperthreading on this payments workload causes a negligible performance degradation for large batches (approximately 1-6%), and a larger (up to 25%) on small batches. As a baseline, §J graphs the performance of Block-STM on these parameter settings on our hardware.

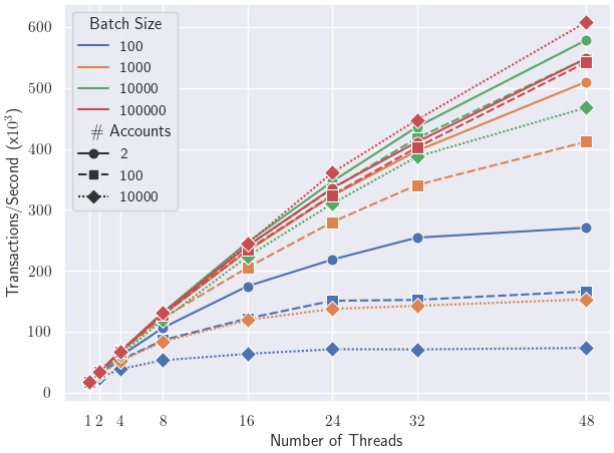

Fig. 7. *Throughput of SPEEDEX on batches of payment transactions with varying thread counts (average of 100 trials).*

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

Production Systems. Finally, we ran the Ethereum Virtual Machine (Geth 1.10 [10]) on a workload of UniswapV2 [24] transactions, and measured a rate of ∼3000 transactions per second (a result in line with other Ethereum benchmarks [107]). The Loopring exchange, built as an L2 rollup on Ethereum, claims a maximum rate of ∼2000 per second [23], a number calculated from Ethereum's per-block computation limit [21], which is in turn set based on the real computational cost of serial transaction execution [26, 45, 47, 91]. Precise measurements of the Stellar blockchain's orderbook DEX suggest that its implementation could handle ∼4000 DEX trades per second.

8 Design Limitations and Mitigations

Latency. Batch trading inherently introduces latency (between order submission and order execution) not present on traditional, centralized exchanges, simply because an order cannot execute until a batch has been closed and clearing prices have been computed. This latency is already present in a blockchain context (a transaction is not finalized until

the consensus protocol adds it to a block), so in this context, SPEEDEX introduces no additional latency.

The latency may have downstream economic effects. Market-making may be more (or less) profitable operating in a batch system, which could lead to reduced (or increased) liquidity. Budish et al. [30] argue that batch trading (between 2 assets) would reduce costs for market-makers, which could lead to increased liquidity. However, they study a higher batch frequency (approximately once per millisecond); our lower batch frequency is less studied (see Q9, [41]).

Tâtonnement Nondeterminism. The algorithms evaluated in §6 can be viewed as a randomized approximation scheme, which raises the question of whether a malicious operator can manipulate the approximation. Note that the level of approximation error (as defined in §B) can be measured, so non-anonymous node operators can be penalized for malfeasance. When regulation is not possible, Tâtonnement can be made deterministic by fixing a set of control parameters for each instance and choosing the solution with the lowest approximation error (or lowest unrealized utility, §6.2). The Stellar implementation uses a static set of control parameters with one Tâtonnement instance. Node operators could also compete to compute prices accurately, as in [7].

Nondeterministic Overdraft Prevention. SPEEDEX needs to prevent an account from spending more than its balance of an asset. As discussed in §3, our implementation considers a proposal valid only if no account is overdrafted after applying the block. This design complicates pipelining of consensus with execution, gives plausible deniability for delaying transactions, and is incompatible with cryptographic commit-reveal schemes.

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

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

Other Types of Front-Running. The set of pending transactions is public in many blockchains. One might estimate the clearing prices in a future batch and arbitrage the batch against low-latency markets. This could lead to negative externalities (see [42], footnote 1), and could merit combining SPEEDEX with a commit-reveal scheme such as [52, 117]. Such a design requires the deterministic overdraft-prevention scheme above.

Malicious nodes might also delay transactions. An implementation could buffer several blocks of transactions from a consensus protocol into a single SPEEDEX batch. If even one of these consensus blocks is from an honest replica (that does not censor transactions), a user could ensure that their transaction cannot be delayed from one SPEEDEX batch to the next (by broadcasting to all replicas). This requires a consensus protocol with sufficient *chain quality* [67]. Alternatively, some DAG-based protocols [56, 79] simultaneously commit many blocks of transactions from different replicas. Grouping these blocks into one SPEEDEX batch, instead of ordering them arbitrarily, achieves the same censorship-resistance property. These designs would likewise require the deterministic overdraft-prevention scheme.

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

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

Limited Trade Types. Trades on SPEEDEX are limited to trades selling a fixed amount of one asset in exchange for as much as possible of another. SPEEDEX does not implement offers to buy a fixed amount of an asset in exchange for as little as possible of another. These buy offers admit the same logarithmic transformation as in §5.1, but make the price computation problem PPAD-hard, a complexity class that is widely conjectured to be algorithmically intractable in polynomial time (§H). One could compute prices using only sell offers and integrate buy offers in the linear programming step.

Ramseyer et al. [96] show how to integrate Constant Function Market Makers (CFMMs) [28] into the exchange market framework and Tâtonnement. The Stellar implementation uses this integration with its own CFMMs.

9 Implementation Details

The standalone SPEEDEX evaluated in §6 and §7 is a blockchain using HotStuff [115] for consensus. A leader node periodically mints a new block from the memory pool and

feeds the block to the consensus algorithm. Other nodes apply the block once it has been finalized by consensus. A faulty node can propose an invalid block. Consensus may finalize invalid blocks, but these blocks have no effect when applied.

The implementation is available open source at https://github.com/scslab/speedex and consists of ∼30,000 lines of C++20, plus ∼5,000 lines for our Hotstuff implementation. It uses Intel's TBB library [8] to manage parallel work scheduling, the GNU Linear Programming Kit [86] to solve linear programs, and LMDB [50] to manage data persistence (for crash recovery).

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

The rest of this section outlines additional design choices built into SPEEDEX. Additional design choices in §K. All optimizations (save §9.1) are implemented in the evaluated system.

9.1 Blockchain Integration

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

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

9.2 Caches and Tâtonnement

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

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

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

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

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

9.3 Batched Trie Design

Our tries use a fan-out of 16 and hash nodes with the 32-byte BLAKE2b cryptographic hash [34]. Both the layout of trie nodes and the work partitioning are designed to avoid having multiple threads writing to the same cache line.

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

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

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

10 Related Work

Blockchain Scaling. Our approach is inspired by Clements et al. [51], who improve performance in the Linux kernel through commutative syscall semantics.

Chen et al. [49] speculatively execute Ethereum transactions

to achieve a ∼6x overall execution speedup. Other approaches to concurrent execution include optimistic concurrency control [70, 111], invalidating conflicting transactions [27], broadcasting conflict resolution information [29, 60], or partitioning transactions into nonconflicting sets [35, 74, 116]. This problem is related to that of building deterministic databases and software transactional memory [94, 105, 110]. Li et al. [83] build a distributed database where some transactions are tagged as commutative.

Empirical work [66, 98] finds that a small number of Ethereum contracts, often token contracts, are historically responsible for the majority of conflicts that limit optimistic execution. A recent Solana [112] outage resulted in part when many transactions conflicted on one orderbook contract [99].

Project Hamilton [85] develops a CBDC payments platform. The authors find that totally-ordered semantics become a performance bottleneck. Unlike SPEEDEX, which stores asset balances in accounts, this system requires the more restrictive unspent transaction output (UTXO) model.

Some systems move transaction execution off-chain, into so-called "Layer-2" networks, each with different capabilities, perfomance, interoperability, and security tradeoffs [11, 13, 18, 21, 78, 92, 93]. Other blockchains [6, 27, 103, 109, 118] split state into concurrently-running shards, at the cost of complicating cross-shard transactions.

(Distributed) Exchanges. Budish et al. [42, 43] argue that exchanges should process orders in batches to combat automated arbitrage and improve liquidity.

Other defenses against front-running include cryptographic commit-reveal schemes [52, 72, 100, 117] or "fair" ordering schemes that assume a bounded fraction of malicious nodes [40, 80, 119]. The front-running attacks that SPEEDEX prevents are not guaranteed to be blocked in these schemes. For example, a replica might plausibly front-run a transaction in [80] by investing in lower-latency network links between itself and other replicas than other replicas have with each other, and commit-reveal schemes do not prevent statistical front-running (guessing the contents of a transaction).

Some blockchains build limit-order DEX mechanisms natively [2, 16] or as smart contracts [14]. Smart contracts known as Automated Market-Makers (AMMs) [24, 64, 73, 87] facilitate passive market-making on-chain [28].

0x and a past version of Loopring [19, 108] allow settlement on-chain of orders matched off-chain, in pairs or in cycles. StarkEx [15, 36] gives cryptographic tools to prove correctness of an off-chain exchange.

CoWSwap [5, 7] uses mixed-integer programming to clear offers in batches of at most 100 [20]. Solvers compete to produce the best solution. The former Binance DEX [3] computed per-asset-pair prices in each block. The Penumbra DEX uses homomorphic encryption to privately make batch swaps against an AMM, but cannot let users set limit prices [12].

Price Computation. Our algorithms solve instances of the special case of the Arrow-Debreu exchange market [33] where every utility function is linear. Equilibria can be approximated in these markets using combinatorial algorithms such as those of Jain et al. [77] and Devanur et al. [58] and exactly via the ellipsoid method and simultaneous diophantine approximation [76]. Duan et al. [62] construct an exact combinatorial algorithm, which Garg et al. [69] extend to an algorithm with strongly-polynomial running time. Ye [114] gives a path-following interior point method, and Devanur et al. [57] construct a convex program. Codenotti et al. [53, 54] show that a version of the Tâtonnement process [32] converges to an approximate equilibrium in polynomial time. Garg et al. [68] give another algorithm based on demand queries.

11 Conclusion

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

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

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

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 (76412CSII). The Stellar blockchain integration was funded by and performed at the Stellar Development Foundation.

The authors wish to thank the anonymous reviewers and our shepherd Siddhartha Sen for their valuable feedback, and thank CloudLab [63] for providing resources for our experiments.

References

[1] Binance chain docs - fees. https://web.archive.org/web/20200617014623/https://

docs.binance.org/guides/concepts/fees.html. Accessed 10/18/2022.

[2] Binance chain docs - introduction. https://web.archive.org/web/20200616190856/https://docs.binance.org/guides/intro.html. Accessed 10/18/2022.

[3] Binance chain docs - match steps and examples. https://web.archive.org/web/20200617065916/https://docs.binance.org/match-examples.html. Accessed 10/18/2022.

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

[5] Cow protocol overview: The batch auction optimization problem. https://web.archive.org/web/20220614183101/https://docs.cow.fi/off-chain-services/in-depth-solver-specification/the-batch-auction-optimization-problem. Accessed 10/19/2022.

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

[7] An exchange protocol for the decentralized web. https://web.archive.org/web/20220825164405/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/c56235a3c79fbd85771760ca8826b757fb03eb1f/BatchAuctionOptimization/batchauctions.pdf.

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

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

[10] Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum/tree/release/1.10. Accessed 10/13/2022.

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

[12] The penumbra protocol: Sealed-bid batch swaps. https://web.archive.org/web/20220614034906/https://protocol.penumbra.zone/main/zswap/swap.html. Accessed 10/19/2022.

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

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

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

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

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

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

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

[20] Gpv2 objective criterion. https://web.archive.org/web/20211019155516/https://forum.gnosis.io/t/gpv2-objective-criterion/1254, April 2021. Accessed 04/30/2021.

[21] 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.

[22] The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure. https://web.archive.org/web/20221020032330/https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf, August 2022. Accessed 10/20/22.

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

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

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

[26] 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.

[27] 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.

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

[29] 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.

[30] Matteo Aquilina, Eric B Budish, and Peter O'Neill. Quantifying the high-frequency trading" arms race": A simple new methodology and estimates. Technical report, Working Paper, 2020.

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

[32] 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.

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

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

[35] 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.

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

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

[38] 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.

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

[40] 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.

[41] Eric Budish. Response to esma's call for evidence: "periodic auctions for equity instruments" (esma70-156-785). https://ericbudish.org/wp-content/uploads/2022/03/response_esmas_call_evidence_periodic_auctions.pdf, January 2019. Accessed 10/17/2022.

[42] Eric Budish, Peter Cramton, and John Shim. Implementation details for frequent batch auctions: Slowing down markets to the blink of an eye. *American Economic Review*, 104(5):418–24, 2014.

[43] 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.

[44] Eric B Budish, Peter Cramton, Albert S Kyle, and Jeongmin Lee. Flow trading. *University of Chicago, Becker Friedman Institute for Economics Working Paper*, (2022-82), 2022.

[45] Vitalik Buterin. A quick explanation of what the point of the eip 2929 gas cost increases in berlin is. https://web.archive.org/web/20211017034159/https://www.reddit.com/r/ethereum/comments/mrl5wg/a_quick_explanation_of_what_the_point_of_the_eip/, April 2021.

[46] 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.

[47] 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.

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

[49] 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.

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

[51] 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.

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

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

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

[61] 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.

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

[63] 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.

[64] Michael Egorov. Stableswap-efficient mechanism for stablecoin liquidity. *Retrieved Feb*, 24:2021, 2019.

[65] Stellar Development Foundation. Stellar for cbdcs. https://resources.stellar.org/hubfs/StellarCBDCWhitepaper.pdf. Accessed 2/24/2023.

[66] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts. *arXiv preprint arXiv:2201.03749*, 2022.

[67] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 281–310. Springer, 2015.

[68] Jugal Garg, Edin Husić, and László A Végh. Auction algorithms for market equilibrium with weak gross substitute demands and their applications. *arXiv preprint arXiv:1908.07948*, 2019.

[69] 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.

[70] 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.

[71] 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.

[72] Christopher Goes, Awa Sun Yin, and Adrian Brink. Anoma: Undefining money. 2021.

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

[74] Graydon Hoare. Core advancement protocol 53: Smart contract data, Mar 2022. https://github.com/stellar/stellar-protocol/blob/master/core/cap-0053.md.

[75] BIS Innovation Hub. Project helvetia phase ii: Settling tokenised assets in wholesale cbdc. https://www.bis.org/publ/othp45.pdf, 2022. Accessed 2/24/2023.

[76] 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.

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

[78] 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.

[79] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

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

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

[82] 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.

[83] 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.

[84] 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.

[85] 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.

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

[87] Fernando Martinelli and Nikolai Mushegian. Balancer whitepaper. Technical report, 9 2019. Accessed 2/4/2022.

[88] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32, 2015.

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

[90] 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.

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

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

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

[94] Guna Prasaad, Alvin Cheung, and Dan Suciu. Handling highly contended oltp workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 527–542, 2020.

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

[96] Geoffrey Ramseyer, Mohak Goyal, Ashish Goel, and David Mazières. Batch exchanges with constant function market makers: Axioms, equilibria, and computation. *arXiv preprint arXiv:2210.04929*, 2022.

[97] Daniël Reijsbergen and Tien Tuan Anh Dinh. On exploiting transaction concurrency to speed up blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1044–1054. IEEE, 2020.

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

[99] 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.

[100] Noah Schmid, Christian Cachin, Orestis Alpos, and Giorgia Marson. Secure causal atomic broadcast, 2021.

[101] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[102] 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.

[103] NEAR Team. Near launches nightshade sharding, paving the way for mass adoption. https://web.archive.org/web/20221007081239/https://near.org/blog/near-launches-nightshade-sharding-paving-the-way-for-mass-adoption/, November 2021. Accessed 10/18/2022.

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

[105] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[106] 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.

[107] 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.

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

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

[110] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: a lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

[111] 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.

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

[113] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, Renton, WA, April 2022. USENIX Association.

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

[115] 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.

[116] 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.

[117] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galinanes, and Bryan Ford. Flash freezing flash boys: Countering blockchain front-running. In *The Workshop on Decentralized Internet, Networks, Protocols, and Systems (DINPS)*, 2022.

[118] 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.

[119] 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 relies on a correspondence between a batch of trade offers and an instance of a linear Arrow-Debreu Exchange Market [33]. Specifically, 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 model from the economics and theoretical computer science literature. Conceptually, there exists in this market a set of independent *agents*, each with its own *endowment* of goods. Each agent has some set of preferences over possible collections of goods. These goods are tradeable on an open market, and agents, all at the same time, make any set of trades that they wish with *the market* (or *auctioneer*), not directly with each other.

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

When the market trades at prices $p \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$, every agent sells their endowment to the market in exchange for revenue $s_j = p \cdot e_j$, which the agent immediately spends at the market to buy back an optimal bundle of goods $x_j \in \mathbb{R}_{\geq 0}^{|\mathfrak{A}|}$ - that is, $x_j = \text{argmax}_{x:\sum_{\mathcal{A}\in\mathfrak{A}} x_{\mathcal{A}} p_{\mathcal{A}} \leq s_j} u_j(x)$.

There are countless variants on this definition. Typically the utility functions are assumed to be quasi-convex. Some variants include stock dividents, corporations, production of new goods from existing goods, and multiple trading rounds. SPEEDEX uses only the model outlined above—SPEEDEX looks only at snapshots of the market, i.e., once per block, and computes batch results for each block independently.

One potential objection to the above definition is that it assumes that the abstract market has sufficient quantities available so that every agent can make its preferred trades. We say that a market is at *equilibrium* when agents can make their preferred trades and the market does not have a deficit in any good.

Definition 2 (Market Equilibrium)**.** *An equilibrium of an Arrow-Debreu market is a set of prices p and an allocation x_j for every agent j, such that for all goods $\mathcal{A}$, $\sum_j e_{\mathcal{A},j} \geq \sum_j x_{\mathcal{A},j}$, and x_j is an optimal bundle for agent j. The inequality for asset $\mathcal{A}$ is tight whenever $p_{\mathcal{A}}$ is nonzero.*

Note that an equilibrium includes both a set of market prices and a choice of a utility-maximizing set of goods for each agent. Say, for example, there are two goods $\mathcal{A}$ and $\mathcal{B}$, and one unit of each is sold by other agents to the market. If two agents are indifferent to receiving either good, then the equilibrium must specify whether the first receives $\mathcal{A}$ or $\mathcal{B}$, and vice versa for the second. It would not be a market equilibrium for both of these agents to purchase a unit of $\mathcal{A}$ and no units of $\mathcal{B}$.

A.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 ($\mathcal{S}$, $\mathcal{B}$, e, α) is request to sell e units of good $\mathcal{S}$ in exchange for some number k units of good $\mathcal{B}$, subject to the condition that $k \geq \alpha e$.*

The user who submits this offer implicitly says that they value k units of $\mathcal{B}$ more than e units of $\mathcal{S}$ if and only if $k \geq \alpha e$. These preferences are representable as a linear utility function.

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

Proof. Such an offer makes no trades if $p_{\mathcal{S}}/p_{\mathcal{B}} < \alpha$ and trades in full if $p_{\mathcal{S}}/p_{\mathcal{B}} > \alpha$.

The user starts with k units of $\mathcal{S}$. In the exchange market model, the user can trade these k units of $\mathcal{S}$ in exchange for any quantities $x_{\mathcal{S}}$ of $\mathcal{S}$ and $x_{\mathcal{B}}$ of $\mathcal{B}$, subject to the constraint that $p_{\mathcal{S}} x_{\mathcal{S}} + p_{\mathcal{B}} x_{\mathcal{B}} \leq k p_{\mathcal{S}}$.

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

Note that these utility functions have nonzero marginal utility for only two types of assets, and are not arbitrary linear utilities. Ramseyer et al. [96] find anecdotal evidence that this subclass of utility functions may be analytically more tractable than the case of general linear utilities.

A.3 Existence of 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. [57]. Existence of an equilibrium with nonzero prices follows therefore from Theorem 1 of [57]. □

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 $\mathcal{A}$ and $\mathcal{B}$ for which $p_\mathcal{A}/p_\mathcal{B} < p'_\mathcal{A}/p'_\mathcal{B}$.*

Then it must be the case that there is a partitioning of the assets $\mathfrak{A}_1, \mathfrak{A}_2$ with $A \in \mathfrak{A}_1, B \in \mathfrak{A}_2$ such that both equilibria include no trading activity across the partition.

Proof. Consider the set of offers trading from $\mathcal{A}$ to $\mathcal{B}$. Let $Z_{\mathcal{A},\mathcal{B}}(r)$ be the set of amounts of asset $\mathcal{A}$ that may be sold (when every agent receives an optimal bundle) by these offers to the market at an exchange rate $r = p_\mathcal{A}/p_\mathcal{B}$. Observe that if $r_1 < r_2$, then every $z_1 \in Z_{\mathcal{A},\mathcal{B}}(r_1)$ is no more than than any $z_2 \in Z_{\mathcal{A},\mathcal{B}}(r_2)$ (as sell offers always prefer higher exchange rates).

At the equilibrium (p,x), let $z_{\mathcal{A},\mathcal{B}}$ be the total amount of $\mathcal{A}$ sold for $\mathcal{B}$ for every asset pair (and $z'_{\mathcal{A},\mathcal{B}}$ similarly for (p',x')). Note that $z_{\mathcal{A},\mathcal{B}} \in Z_{\mathcal{A},\mathcal{B}}(p_\mathcal{A}/p_\mathcal{B})$.

Suppose that there exists a pair of assets $\mathcal{A}, \mathcal{B}$ as in the theorem statement. Then there exists a set of assets $\mathfrak{A}_1$ such that for every asset pair $\mathcal{C} \in \mathfrak{A}_1$ and $\mathcal{D} \notin \mathfrak{A}_1$, $p_\mathcal{C}/p_\mathcal{D} < p'_\mathcal{C}/p'_\mathcal{D}$.

For each of these asset pairs, we must have that $z_{\mathcal{C},\mathcal{D}} \leq z'_{\mathcal{C},\mathcal{D}}$, $z_{\mathcal{D},\mathcal{C}} \geq z'_{\mathcal{D},\mathcal{C}}$, and $\frac{p_\mathcal{C}}{p_\mathcal{D}} z_{\mathcal{C},\mathcal{D}} \leq \frac{p'_\mathcal{C}}{p'_\mathcal{D}} z'_{\mathcal{C},\mathcal{D}}$. Combining these equations gives

$$p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq (p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}}) p_\mathcal{D}/p'_\mathcal{D}$$

Each of these inequalities is tight if and only if $z_{\mathcal{C},\mathcal{D}} = 0$.

It is without loss of generality to rescale p' so that $p_\mathcal{D}/p'_\mathcal{D} < 1$ for all $\mathcal{D} \notin \mathfrak{A}_1$. Thus,

$$p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \leq (p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}})$$

Because (p,x) and (p',x') are equilibria, we must have that

$$\begin{aligned} 0 &= \sum_{\mathcal{C} \in \mathfrak{A}_1} \sum_{\mathcal{D} \notin \mathfrak{A}_1} p_\mathcal{C} z_{\mathcal{C},\mathcal{D}} - p_\mathcal{D} z_{\mathcal{D},\mathcal{C}} \\ &\leq \sum_{\mathcal{C} \in \mathfrak{A}_1} \sum_{\mathcal{D} \notin \mathfrak{A}_1} p'_\mathcal{C} z'_{\mathcal{C},\mathcal{D}} - p'_\mathcal{D} z'_{\mathcal{D},\mathcal{C}} \end{aligned}$$

But the second inequality is tight only if each $z_{\mathcal{C},\mathcal{D}} = 0$.

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

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

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

If G is connected, then the market equilibrium prices p are unique (up to uniform rescaling).

Proof. If the theorem hypothesis holds, then for any other equilibrium (p',x'), it must be the case that for every asset pair $(\mathcal{A},\mathcal{B})$, $p_\mathcal{A}/p_\mathcal{B} = p'_\mathcal{A}/p'_\mathcal{B}$. By Theorem 4, if this did not hold, then there would exist a partitioning of V into two sets of assets, across which there is no trading at equilibrium (p,x) (contradicting the assumption that G is connected). □

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 $p_\mathcal{A}$ on each asset $\mathcal{A}$, and a trade amount $x_{\mathcal{A},\mathcal{B}}$ denoting the amount of $\mathcal{A}$ sold in exchange for $\mathcal{B}$.

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

1. Asset conservation is preserved with an ε commission. The amount of $\mathcal{A}$ sold to the auctioneer, $\Sigma_\mathcal{B} x_{\mathcal{A},\mathcal{B}}$, must exceed the amount of $\mathcal{A}$ bought from the auctioneer, $\Sigma_\mathcal{B}(1-\varepsilon)\frac{p_\mathcal{B}}{p_\mathcal{A}} x_{\mathcal{B},\mathcal{A}}$.

2. No offer trades outside of its limit price. That is to say, an offer selling $\mathcal{A}$ for $\mathcal{B}$ with a limit price of r cannot execute if $\frac{p_\mathcal{A}}{p_\mathcal{B}} < r$.

3. No offer with a limit price "far" from the batch exchange rate does not trade. That is to say, an offer selling $\mathcal{A}$ for $\mathcal{B}$ with a limit price of r must trade in full if $r < (1-\mu)\frac{p_\mathcal{A}}{p_\mathcal{B}}$.

   Intuitively, the lower the limit price, the more an offer prefers trading to not trading.

This notion of approximation is closely related to but not exactly the same as notions of approximation used in the theoretical literature on Arrow-Debreu exchange markets (e.g., [53], 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. [53] is the method in which Tâtonnement adjusts prices in response to a demand query. Codenotti et al. use an additive rule that they find amenable to theoretical analysis. If $Z(p)$ is the market demand at prices p, they update prices according to the following rule:

$$p_\mathcal{A} \leftarrow p_\mathcal{A} + Z_\mathcal{A}(p)\delta \tag{1}$$

for some constant δ. The authors show that there is a sufficiently small δ so that Tâtonnement is guaranteed to move closer to an equilibrium after each step.

The relevant constant is unfortunately far too small to be usable in practice, and more generally, we want an algorithm that can quickly adapt to a wide variety of market conditions (not one that always proceeds at a slow pace).

First, we update prices multiplicatively, rather than additively. This dramatically reduces the number of required rounds, especially when Tâtonnement starts at prices that are far from the clearing prices.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+Z_{\mathcal{A}}(p)\delta) \tag{2}$$

Second, we normalize asset amounts by asset prices, so that our algorithm will be invariant to redenominating an asset. It is equivalent to trade 100 pennies or 1 USD, and our algorithm performs better when it can take that kind of context into account.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+p_{\mathcal{A}}Z_{\mathcal{A}}(p)\delta) \tag{3}$$

Next, we make δ a variable factor. We use a heuristic to guide the dynamic adjustment. Our experiments used the l^2 norm of the price-normalized demand vector, $\sum_{\mathcal{A}}(p_{\mathcal{A}}Z_{\mathcal{A}}(p))^2$; other natural heuristics (i.e. other l^p norms) perform comparably (albeit not quite as well). In every round, Tâtonnement computes this heuristic at its current set of candidate prices, and at the prices to which it would move should it take a step with the current step size. If the heuristic goes down, Tâtonnement makes the step and increases the step size, and otherwise decreases the step size. This is akin to a backtracking line search [31, 39] with a weakened termination condition.

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+p_{\mathcal{A}}Z_{\mathcal{A}}(p)\delta_t) \tag{4}$$

Finally, we normalize adjustments by a trade volume factor $\nu_{\mathcal{A}}$. Without this adjustment factor, computing prices when one asset is traded much less than another asset takes a large number of rounds, simply because the lesser traded asset's price updates are always of a lower magnitude than those of the more traded asset. Many other numerical optimization problems run most quickly when gradients are normalized (e.g., see [37]).

$\nu_{\mathcal{A}}$ need not be perfectly accurate—indeed, knowing the factor exactly would require first computing clearing prices—but we can estimate it well enough from the trading volume in prior blocks and from trading volume in earlier rounds of Tâtonnement (specifically, we use the minimum of the amount of an asset sold to the auctioneer and the amount bought from the auctioneer). Real-world deployments could estimate these factors using external market data.

Putting everything together gives the following update rule:

$$p_{\mathcal{A}} \leftarrow p_{\mathcal{A}}(1+p_{\mathcal{A}}Z_{\mathcal{A}}(p)\delta_t\nu_{\mathcal{A}}) \tag{5}$$

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., [39]). Devanur et al. [57] even give a convex objective function for computing exchange market equilibria, which we reproduce below (in a simplified form):

$$\sum_{i:mp_i<\frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}} p_{\mathcal{S}_i}E_i\ln\left(mp_i\frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}\right) - y_i\ln(mp_i) \tag{6}$$

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

This objective is accompanied by an asset conservation constraint for each asset $\mathcal{A}$:

$$\sum_{i:\mathcal{S}_i=\mathcal{A}} y_i = \sum_{i:\mathcal{B}_i=\mathcal{A}} y_i \tag{7}$$

However, unlike the problem formulation in [57], Tâtonnement does not have decision variables $\{y_i\}$. Rather, Tâtonnement pretends offers respond rationally to market prices, and then adjusts prices so that constraints become satisfied. As such, mapping our algorithms onto the above formulation would mean that $y_i = p_{\mathcal{S}_i}E_i$ if $mp_i < \frac{p_{\mathcal{S}_i}}{p_{\mathcal{B}_i}}$ and 0 otherwise (although §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

$$\sum_{\mathcal{A}}\lambda_{\mathcal{A}}\left(\sum_{i:\mathcal{S}_i=\mathcal{A}} y_i(p) - \sum_{i:\mathcal{B}_i=\mathcal{A}} y_i(p)\right) \tag{8}$$

for a set of langrange multipliers $\{\lambda_{\mathcal{A}}\}$.

We write $y_i(p)$ to denote that in this formulation, offer behavior is directly a function of prices. It appears difficult to use equation 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 $\lambda_{\mathcal{A}}$ is the amount by which (the additive version of) Tâtonnement updates $p_{\mathcal{A}}$. This might suggest using $p_{\mathcal{A}}$ in place of $\lambda_{\mathcal{A}}$ in equation 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 non-differentiable 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 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, SPEEDEX linearly 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 an $\frac{\alpha-\beta}{\mu\alpha}$ fraction of its assets.

Observe that as μ gets increasingly small, this linear interpolation becomes an increasingly close approximation of a step function. This explains some of the behavior in Figure 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 $x_{A,B}$. 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:

$$\sum_B x_{A,B} \geq \sum_B (1-\varepsilon)\frac{p_B}{p_A} x_{B,A}$$

Define $U_{A,B}$ 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 at or below $\frac{p_A}{p_B}$), and define $L_{A,B}$ 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 at or 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),

$$L_{A,B} \leq x_{A,B} \leq U_{A,B}$$

Informally, the goal of our linear program is to maximize the total amount of trading activity. Any measurement of trading activity needs to be invariant to redenominating assets; intuitively, it is the same to trade 1 USD or 100 pennies. As such, the objective of our linear program is:

$$\sum_{A,B} p_A x_{A,B}$$

Putting this all together gives the following linear program (let $\mathfrak{A}$ be the set of all assets):

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

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

$$p_A \sum_{B \in \mathfrak{A}} x_{A,B} \geq (1-\varepsilon) \sum_{B \in \mathfrak{A}} p_B x_{B,A} \quad \forall A \in \mathfrak{A} \tag{11}$$

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 $x_{A,B}$ to be 0 instead of $L_{A,B}$. This change makes the program always feasible (e.g., an assigment of each variable to 0 satisfies the constraints).

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

This substitution gives the following linear program:

$$\max \sum_{A,B} y_{A,B} \tag{12}$$

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

$$\sum_{B \in \mathfrak{A}} y_{A,B} \geq (1-\varepsilon) \sum_{B \in \mathfrak{A}} y_{A,B} \quad \forall A \tag{14}$$

The Stellar implementation charges no transaction commission (i.e., sets ε to 0) in its SPEEDEX deployment. This makes the linear program into an instance of the maximum circulation problem (i.e., variable $y_{A,B}$ denotes the flow from vertex A to vertex B). It is well known that the constraint matrices of these problems are totally unimodular (Chapter 19, Example 4 [101]). This means that it always has an integral solution (Theorem 19.1, [101]) and can be solved by specialized algorithms (such as those outlined in [81]). Some of these algorithms run substantially faster than general simplex-based solvers.

Appendix E Market Structure Decomposition

Suppose that the set of goods could be partitioned between a set of numeraires, which might be traded with any other asset, and a set of stocks, which are only traded with one of the pricing assets.

Then SPEEDEX could compute a batch equilibrium by first computing an equilibrium taking into account only trades between pricing assets, then computing an equilibrium exchange rate for every stock between the stock and its pricing asset, and finally combining the results.

More specifically:

Theorem 5. *Let $\mathfrak{A}$ be the set of numeraires and $\mathfrak{S}$ the set of stocks. A stock $S \in \mathfrak{S}$ is traded with asset $a(S) \in \mathfrak{A}$.*

Suppose (p,x) is an equilibrium for the restricted market instance considering only the numeraires. For each $S \in \mathfrak{S}$, 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* $A \in \mathfrak{A}$
2. $p'_S = (r_S / r_{a(S)}) p_{a(S)}$
3. $x'_{A,B} = x_{A,B}$ *for* $A, B \in \mathfrak{A}$
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). □

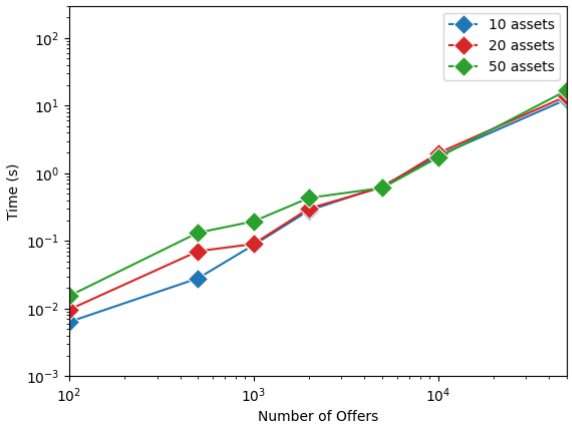

Fig. 8. *Time to solve the convex program of Devanur et al. [57] using the CVXPY toolkit [59], varying the number of assets and offers.*

Appendix F Alternative Batch Solving Strategies

F.1 Convex Optimization

We implemented the convex program of Devanur et al. [57] directly, using the CVXPY toolkit [59] backed by the ECOS convex solver [61]. Figure 8 plots the runtimes we observed to solve the problem while varying the number of assets and offers.

The runtimes are not directly comparable to those of Tâtonnement—namely, this strategy does not have the potential to shortcircuit operation upon early arrival at an equilibrium (our notions of approximation error also do not directly translate to the notions used interally in the solver), nor is it optimized for our particular class of problems.

The important observation is that the runtime of this strategy scales linearly in the number of trade offers. Instances trading 1000 offers, for example, take roughtly 10x as long as instances trading only 100 offers.

This is not a surprising result, given that the number of variables in the convex program scales linearly with the number of trade offers.

The choice of solver strategy does not, of course, change the structure of the input problem instances. The same observation used in §5.1 makes it possible to refactor the convex program so that the number of variables does not depend on the number of open offers, and so that the objective (and its derivatives) can be evaluated in time logarithmic in the number of open offers.

Unfortunately, this transformation makes the objective nondifferentiable. The demand smoothing tactic of §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 [57] while maintaining sufficient smoothness and numerical stability is an interesting open problem.

F.2 Mixed Integer Programming

Gnosis (Walther, [7]) 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 [7] 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 §9.2, Tâtonnement lays out contiguously in memory all the information it needs to return this result quickly.

For a version of Tâtonnement without the demand smoothing of §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)

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

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)

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

We can rearrange the second term of the summation into

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

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

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

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., [53]). 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* $(-50\ EUR, 100\ USD)$.

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

Observe that the price of USD rose and the demand for EUR fell.

Informally speaking, if offers do not satisfy the core logic of Tâtonnement's price update rule, then Tâtonnement cannot handle them in a mathematically sound manner.

More formally, Chen et al. [48] 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 Deterministic Filtering Performance

The deterministic transaction batch pruning system works by eliminating the transactions from all of the accounts that could create an unresolvable conflict. To be specific, if the sum of the amount of an asset used (either sent in a payment option or locked to create a offer) by all of an account's transactions exceeds that account's balance, then that account's transactions are removed. If an account sends two transactions with the same sequence number (both of which have valid signatures, and the sequence numbers are higher than the sequence number of the account's most recent transaction), or two transactions cancel the same offer ID, then that account's transactions are removed. If two transactions create the same account ID, then both transactions are removed.

We generated batches of 400,000 transactions from the same synthetic transaction model as in §7, and then duplicated 100,000 transactions at random to create a batch of 500,000.

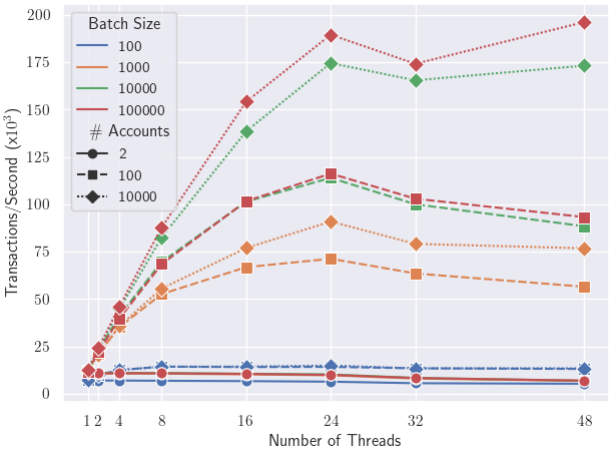

Fig. 9. *Throughput of Block-STM on batches of "Aptos p2p" transactions with varying thread counts (average of 100 trials).*

A small number of accounts (1000) send transactions with conflicting sequence numbers. We initialize the database (again, 10 million accounts) to give each account a small amount of money, and a small number (one or two hundred) of accounts attempt to overdraft.

This filtering takes 0.13s and 0.07s seconds with 24 and 48 threads, respectively (averaged over 50 trials, after a warmup), giving a 21.0× and 38.4× speedup over the serial benchmark. On a more contested benchmark, with only 10,000 accounts (almost all of which overdraft) the maximum speedup over the single threaded trial is only 5.3×, but the overall filtering runtime is still just 0.10s. Our implementation of the filtering is not heavily optimized, but in either parameter setting, the overhead is small.

Appendix J Block-STM Baseline

To provide a baseline for the measurements in Fig. 7, we also ran Block-STM on our hardware (with hyperthreading disabled, as in [70]). Fig. 9 displays the results.

These performance measurements are similar, quantitatively and qualitatively, to those reported in [70] (on different hardware). Note that performance appears to reach a maximum after approximately 16 to 24 threads, and, unlike SPEEDEX, does not effectively use additional hardware beyond this point, even on relatively low-contention workloads.

Appendix K Additional Implementation Details

K.1 Data Organization

Account balances are stored in a Merkle-Patricia trie. However, because a trie is not self-rebalancing, its worst-case adversarial lookup performance can be slow. As such, we store account balances in memory indexed by a red-black tree, with updates pushed to the trie once per block.

For each pair of assets ($\mathcal{A}$, $\mathcal{B}$), we build a trie storing offers selling asset $\mathcal{A}$ in exchange for $\mathcal{B}$. Finally, in each block, we build a trie logging which accounts were modified.

We store information in hashable tries so that nodes can efficiently compare their database state with another replica's (to validate consensus and check for errors), and construct short proofs for users about exchange state.

K.2 Data Storage and Persistence

SPEEDEX uses a combination of an in-memory cache and ACID-compliant databases (several LMDB [50] instances). This choice suffices for our experiments, but a database that persists data in epochs, like Silo [106], or is otherwise optimized for batch operation might improve performance.

Our implementation uses one LMDB instance for the set of open offers, one instance for Hotstuff logs, one instance for storing block headers, and 16 instances for storing account states. LMDB is single-threaded, and we find that the throughput of one thread generating database writes does not keep up with SPEEDEX. Accounts are randomly divided between these instances, according to a hash function keyed by a (persistent) secret key (which is different per blockchain node). This key must be kept secret so as to prevent nodes from denial of service attacks.

Processing transactions in a nondeterministic order complicates recovery from a database snapshot where a block has been partially applied. Cancellation transactions, in particular, refund to an account the remainder of an offer's asset amount. We therefore cannot recover if the snapshot of the orderbooks is more recent than the snapshot of the set of account balances, and our implementation takes care to commit updates to the account LMDB instances before committing updates to the orderbook LMDB.

K.3 Follower Optimizations

A block proposal includes the output of Tâtonnement and the linear program in (the prices and trade amounts, as in §4.2). This permits the nondeterminism in Tâtonnement (§5.2), and lets the other nodes skip the work of running Tâtonnement.

Proposals also include, for every pair of assets, the trie key of the offer with the highest minimum price that trades in that block. When executing a proposal from another node, a follower can compare the trie key of a newly created offer with this marginal key and know immediately whether to make a trade or add the offer to the resting orderbooks. A node also defers all checks that an account balance is not overdrafted to after it has executed all the transactions in a block.

K.4 Replay Prevention

Transactions have per-account sequence numbers to ensure a transaction can execute only once. Many blockchains require sequence numbers from an account to increase strictly sequentially. Our implementation allows small gaps in sequence numbers, but restricts sequence numbers to increase by at most an arbitrary limit (64) in a given block. Allowing

gaps simplifies some clients (such as our open-loop load generator), but more importantly lets validators efficiently track consumed sequence numbers out of order with a fixed-size bitmap and hardware atomics.

The Stellar implementation requires strictly consecutive sequence numbers, mostly for backwards compatibility.

K.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.

K.6 Nondeterministic Block Assembly

As discussed in §3, SPEEDEX must assemble blocks of transactions in a manner that guarantees no account is overdrafted after applying all of the transactions in the block. The block proposal system (Fig. 1, 2) manages this by carefully controlling writes to shared state.

The proposal module takes as input a set of unconfirmed transactions (the "mempool", in typical blockchain parlance) and outputs a proposed block containing a subset of the unconfirmed transactions. For each candidate unconfirmed transaction, a thread reserves the ability to perform all necessary modifications by "locking" all relevant data elements. Once a transaction acquires all of its locks, it performs its necessary state modifications and finally releases the locks. If it cannot acquire all necessary locks, it releases any locks and excludes the transaction from the proposed block.

Conceptually, a transaction offering a trade or sending a payment must lock the number of units of assets that could be debited from the account if the operation succeeds. However, doing this with spinlocks would preclude the scalability displayed in Figure 7. Instead, most reservations are performed with hardware atomics to decrement the number of available units. Crediting an account can never fail because SPEEDEX caps the total amount of any asset issued at `INT64_MAX`. This process is conservative in that it may reject transactions that could have executed safely.

Unique offer IDs ensure that no offer is created twice, and atomic boolean flags ensure an offer cannot be cancelled twice. Sequence numbers can be reserved by atomic bitmaps (as in §K.4). For simplicity, our implementation does use exclusive locks when creating new accounts (which we assume occurs relatively infrequently).

Appendix L Additional Replicas

SPEEDEX invokes a consensus protocol no more than once per second in our experiments. To demonstrate that this overhead is negligible, we ran SPEEDEX with 10 replicas, although with weaker hardware per replica, due to resource limitations. Each replica is one AWS c5ad.16xlarge instance, with one AMD EPYC 7R32 processor (48 CPUs @ 2.8Ghz per physical chip, 32 of which are allocated to our instances), 128 GB of memory, and two 1.1TB NVMe drives in a RAID0 configuration. Performance measurements are plotted in Figure 10.

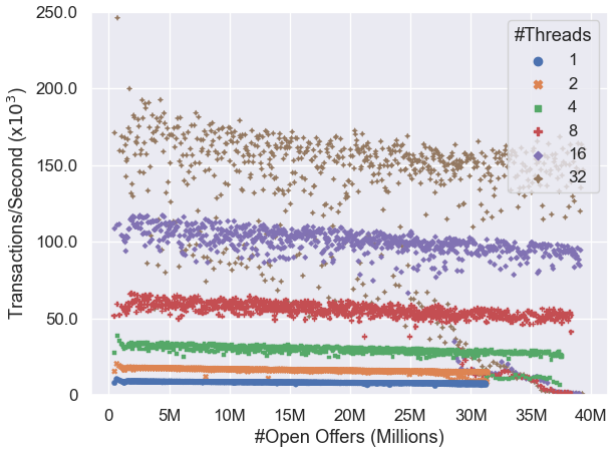

Fig. 10. *Transactions per second on SPEEDEX when running with 10 replicas (on weaker hardware than in Fig. 3), plotted over the number of open offers.*

The overall throughput numbers are lower here than in Figure 3 due to the weaker hardware, but the scalability trends are the same. Doubling the thread count increases performance by a factor of between 1.8x and 1.9x, except that the jump from 16 to 32 gives a roughly 1.4x increase due to contention with background tasks (particularly logging to persistent storage).

This graph also highlights how SPEEDEX responds to insufficient hardware resources. As the number of open offers increases, SPEEDEX's memory requirements increase. Eventually, memory starts to be paged to disk, which dramatically increases disk usage and contends with the logging to persistent storage. SPEEDEX slows down in response, to ensure for safety that data in peristent storage is never too far out of sync.