Designing Multidimensional Blockchain Fee Markets

Theo Diamandis
MIT CSAIL, Cambridge, MA, USA

Alex Evans
Bain Capital Crypto, San Francisco, CA, USA

Tarun Chitra
Gauntlet, New York, NY, USA

Guillermo Angeris
Bain Capital Crypto, San Francisco, CA, USA

Abstract

Public blockchains implement a fee mechanism to allocate scarce computational resources across competing transactions. Most existing fee market designs utilize a joint, fungible unit of account (*e.g.*, *gas* in Ethereum) to price otherwise non-fungible resources such as bandwidth, computation, and storage, by hardcoding their relative prices. Fixing the relative price of each resource in this way inhibits granular price discovery, limiting scalability and opening up the possibility of denial-of-service attacks. As a result, many prominent networks such as Ethereum and Solana have proposed multidimensional fee markets. In this paper, we provide a principled way to design fee markets that efficiently price multiple non-fungible resources. Starting from a loss function specified by the network designer, we show how to dynamically compute prices that align the network's incentives (to minimize the loss) with those of the users and miners (to maximize their welfare), even as demand for these resources changes. We derive an EIP-1559-like mechanism from first principles as an example. Our pricing mechanism follows from a natural decomposition of the network designer's problem into two parts that are related to each other via the resource prices. These results can be used to efficiently set fees in order to improve network performance.

2012 ACM Subject Classification Theory of computation → Convex optimization; Information systems → Digital cash; Theory of computation → Algorithmic mechanism design

Keywords and phrases Blockchains, transaction fees, convex optimization, mechanism design

Digital Object Identifier 10.4230/LIPIcs.AFT.2023.4

Related Version *Full Version*: `https://arxiv.org/abs/2208.07919` [27]

Supplementary Material *Software (Source Code)*: `https://github.com/bcc-research/resource-pricing`, archived at `swh:1:dir:7fda6b4657a257cb9fd944628b905f45ec938c09`

Acknowledgements We would like to thank John Adler, Vitalik Buterin, Dev Ojha, Kshitij Kulkarni, Matheus Ferreira, Barnabé Monnot, and Dinesh Pinto for helpful conversations, insights, and edits. We're especially appreciative to John Adler for bearing with us through many drafts of this work and consistently providing valuable feedback.

1 Introduction

Public blockchains allow any user to submit a *transaction* that modifies the shared state of the network. Transactions are independently verified and executed by a decentralized network of *full nodes*. Because full nodes have finite resources, blockchains limit the total computational resources that can be consumed per unit of time. As user demand may fluctuate, most blockchains implement a *transaction fee* mechanism in order to allocate finite computational capacity among competing transactions.

© Theo Diamandis, Alex Evans, Tarun Chitra, and Guillermo Angeris;
licensed under Creative Commons License CC-BY 4.0

Smart contracts and gas. Many blockchains allow transactions to submit and/or execute programs that exist on-chain called *smart contracts*. Once such a transaction is included in a block, full nodes must re-execute the transaction in order to obtain the resulting updated state of the ledger. All of these transactions consume computational resources, whose total supply is finite. To prevent transactions with excessive resource use and transaction spam, some smart-contract blockchains require users to pay fees in order to compensate the network for processing their transactions.

Most smart contract platforms calculate transaction fees based on a joint unit of account. In the Ethereum Virtual Machine (EVM), this unit is called *gas*. Each operation in the EVM requires a hardcoded amount of gas which is intended to reflect its relative resource usage. The network enforces a limit on the total amount of gas consumed across all transactions in a block. This limit, called the *block limit*, prevents the chain from expending computational resources too quickly for full nodes to catch up to the latest state in a reasonable amount of time. Block limits must take into account the maximum amount of each resource that each block may consume (such as storage, bandwidth, or memory) without posing an extreme burden on full nodes meeting the minimal hardware specifications. Because the block limit fixes the total gas supply in each block, the price of gas in "real" terms (*e.g.*, in terms of US Dollars) fluctuates based on demand for transactions in the block.

One-dimensional transaction fees. Calculating transaction fees through a single, joint unit of account, such as gas, introduces two major challenges. First, if the hardcoded costs of each operation are not precisely reflective of their relative resource usage, there is a possibility of denial-of-service attacks (specifically, *resource exhaustion attacks* [44]), where an attacker exploits resource mispricing to overload the network. Historically, the Ethereum network has suffered from multiple DoS attacks [13, 14, 52] and has had to manually adjust the relative prices accordingly (*e.g.*, [12, 22]). Amending the hardcoded costs of each gas operation in response to such attacks typically requires a hard fork of the client software.

Second, one-dimensional fee markets limit the theoretical throughput of the network. Using a joint unit of account, such as gas, to price separate resources decouples their price from supply and demand. As an extreme example to illustrate this dynamic, if the block gas limit is fully saturated with exclusively CPU-intensive operations, gas price will increase as transactions compete for limited space. The cost of transactions that consume exclusively network bandwidth (and nearly no CPU resources) will also increase because these also require gas, even if demand and supply for bandwidth resources across the network remain unchanged. As a result, fewer bandwidth-intensive transactions can be included in the block despite spare capacity, limiting throughput. This limitation occurs because the joint unit of account only allows the network to price resources *relative to each other* and not in real terms based on the supply and demand for each resource. As we will soon discuss, allowing resources to be priced separately promotes more efficient resource utilization by enabling more precise price discovery. We note that this increase in throughput need not increase hardware requirements for full nodes.

Multidimensional fee markets. Due to the potential scalability benefits of more granular price discovery, a number of popular smart contract platforms are actively exploring multidimensional fee market mechanisms [5, 18]. We discuss some example proposals that are under active development below.

Rollups and data markets. Rollups are a popular scaling technology that effectively decouples transaction validation and execution from data and consensus [17]. In rollups, raw transaction data is posted to a base blockchain. Rollups also periodically post succinct proofs of valid execution to the base chain in order to enable secure bridging, prevent rollbacks, and arbitrate potential disputes by using the base chain as an anchor of trust. Rollups naturally create two separate fee markets, one for base layer transactions and one for rollup execution. As rollups have become a popular design pattern for achieving scalability, specialized blockchains (called *lazy blockchains*) that exclusively order raw data through consensus (*i.e.*, do not perform execution) have emerged [8, 7, 45, 43, 51]. These blockchains naturally allow for transaction data/bandwidth and execution to be priced through independent (usually one-dimensional) fee markets [4]. Similarly, Ethereum developers have proposed EIP-2242, wherein users may submit special transactions which contain an additional piece of data called a *blob* [2, 1]. Blobs may contain arbitrary data intended to be interpreted and executed by rollups rather than the base chain. Later, EIP-4844 extended these ideas by establishing a two-dimensional fee market wherein data blobs and base-chain gas have different limits and are priced separately [19]. EIP-4844 therefore intends to increase scalability for rollups, as blobs do not have to compete with base-chain execution.

Incentivizing parallelization. Most smart contract platforms, including the EVM, execute program operations sequentially by default, limiting performance. There are several proposals to enable parallel execution in the EVM which generally fall into two categories. The first involves minimal changes to the EVM and pushes the responsibility of identifying opportunities for parallel execution to full nodes [33, 24, 49]. The other approach involves *access lists* which require users to specify which accounts their transaction will interact with, allowing the network to easily identify non-conflicting transactions that can be executed in parallel [15, 23]. While Ethereum makes access lists optional, other virtual machine implementations, such as Solana Sealevel and FuelVM, require users to specify the accounts their transaction will interact with [53, 38, 3]. Despite this capability, a large fraction of transactions often want to access the same accounts in scenarios including auctions, arbitrage opportunities, and new product launches. Such contention significantly limits the potential benefits of the virtual machine's parallelization capabilities. As a result, developers of Solana proposed multidimensional fee markets that price interactions with each account separately [54]. Transactions which require sequential execution are charged higher fees, incentivizing usage of spare capacity on multi-core machines.

This paper. In this paper, we formally illustrate how to efficiently update resource prices, what optimization problem these updates attempt to solve, and some consequences of these observations. We frame the pricing problem in terms of an idealized, omniscient network designer who chooses transactions to include in blocks in order to maximize total welfare, subject to demand constraints. (The designer is omniscient as the welfare is unknown and likely unmeasurable in any practical setting.) We show that this problem, which is the "ideal end state" of a blockchain but not immediately useful by itself, decomposes into two problems, coupled by the resource prices. One of these two problems represents the cost to the network for providing certain resources and is simple enough to be solved on chain. The other is a maximal-utility problem that miners and users implicitly solve when creating and including transactions for a given block. Correctly setting the resource prices aligns incentives such that the resource costs to the network are exactly balanced by the utility gained by the users and miners. This, in turn, leads to block allocations which solve the original "ideal" problem, on average.

For convenience, we provide appendix A in the full version [27] as a short introduction to convex optimization. We recommend readers unfamiliar with convex optimization at least skim this appendix, as it provides a short introduction to all the mathematical definitions and major theorems used in this paper.

1.1 Related work

The resource allocation problem has been studied in many fields, including operations research and computer systems. Agrawal, et al. [6] proposed a similar formulation and price update scheme for fungible resources where utility is defined per-transaction. Prior work on blockchain transaction fees varies from the formal axiomatic analysis of game theoretic properties that different fee markets should have [48, 25] to analysis of dynamic fees from a direct algorithmic perspective [31, 39, 47]. Works of the latter form generally focus on whether the system converges to an equilibrium. Moreover, these mechanisms focus on dynamic pricing at the block level (*e.g.*, how many transactions should be allowed in a block?) and not directly on questions of how capacity should be allocated and priced across different transaction types.

EIP-1559. EIP-1559 [21], implemented last year, proposed major changes to Ethereum's transaction fee mechanism. Specifically, EIP-1559 implemented a base fee for transactions to be included in each block, which is dynamically adjusted to hit a target block usage and burned instead of rewarded to the miners. While EIP-1559 is closely related to the problem we consider, it ultimately has a different goal: EIP-1559 attempts to make the fee estimation problem easier in a way that disincentivizes manipulation and collusion [16, 48]. We, on the other hand, aim to price resources dynamically to achieve a given network-specified objective. Finally, we note that prior work such as [31] has proved incentive compatibility for a large set of mechanisms that are a superset of EIP-1559. It is likely (but not proven in this work) that our model fits within an extension of their incentive compatibility framework. We leave game theoretic analysis and strategies to ensure incentive compatibility as future work.

2 Transactions and resources

Before introducing the network's resource pricing problem, we discuss the general set up and motivation for the problem in the case of blockchains.

Transactions. We will start by reasoning about *transactions.* A transaction can represent arbitrary data sent over the peer-to-peer network in order to be appended to the chain. Typically, a transaction will represent a value transfer or a call to a smart contract that exists on chain. These transactions are broadcasted by users through the peer-to-peer network and collected by consensus nodes in the *mempool*, which contains all submitted transactions that have not been included on chain. A *miner* gets to choose which transactions from the mempool are included on chain. Miners may also outsource this process to a block builder in exchange for a reward [20]. Once a transaction is included on chain, it is removed from the mempool. (Any conflicting transactions are also removed from the mempool.)

Nodes. Every transaction needs to be executed by *full nodes* (which we will refer to as *nodes*). Nodes compute the current state of the chain by executing and checking the validity of all transactions that have been included on the chain since the genesis block. Many blockchains have minimum computational requirements for nodes in a blockchain: any node meeting these requirements should be able to eventually "catch up" to the head of the chain in a reasonable amount of time, *i.e.*, execute all transactions and reach the latest state, even as the chain continues to grow. (For example, Ethereum requires 4GB RAM and 2TB of SSD Storage, and they recommend at least a Intel NUC, 7th gen processor [30].) These requirements both limit the computational resources each block is allowed to consume and promote decentralization by ensuring the required hardware does not become prohibitively expensive. If transactions are included in a blockchain faster than nodes are able to execute them, nodes cannot reconstruct the latest state and can't ascertain the validity of the chain. This type of denial of service (DoS) attack is also referred to as a resource exhaustion attack. (As a side note, in some systems, it is possible to provide an easily-verifiable certificate that the state is correct. This certificate allows nodes to validate the state of the chain without executing all past transactions. In these systems, the time-consuming step is generating the certificate. A similar market mechanism might make sense for these systems, but we do not explore this topic here.)

Resource targets and limits. There are several ways to prevent this type of denial of service attack. For example, one method is to enforce that any valid transaction (or sequence of transactions, *e.g.*, a block) consumes at most some fixed upper bound of resources, or combinations of resources. These limits are set so that a node satisfying the minimum node requirements is always able to process transactions quickly enough to catch up to the head of the chain in a reasonable amount of time. Another possibility is to disincentivize miners and users from repeatedly including transactions that consume large amounts of resources while allowing short "bursts" of resource-heavy transactions. This margin needs to be carefully balanced so that a node meeting the minimum requirements is able to catch up after a certain period of time. This intuition suggests having both a "resource target" and a larger "resource limit," which we will formalize in what follows.

Resources. Most blockchain implementations define a number of *meterable resources* (simply *resources* from here on out) which we will label $i = 1, \dots, m$, that a transaction can consume. For example, in Ethereum, the resources could be the individual Ethereum Virtual Machine (EVM) opcodes used in the execution of a transaction. In this paper, the notion of a resource is much more general than simply an opcode or an "execution unit". Here, a resource can refer to anything as coarse as "total bandwidth usage" or as granular as individual instructions executed on a specific core of a machine – the only requirement for a resource, as used in this paper, is that it can be easily and consistently metered across any node. For a given transaction $j = 1, \dots, n$, we will let $a_j \in \mathbb{R}^m_+$ be the vector of resources that transaction j consumes. In particular, the ith entry of this vector, $(a_j)_i$, denotes the amount of resource i that transaction j uses. We note that the resource usage $(a_j)_i$ does not, in fact, need to be nonnegative. While our mechanism works in the more general case (with some small modifications), we assume nonnegativity in this work for simplicity.

Combined resources. This framework naturally includes combinations of resources as well. For example, we may have two resources R_1 and R_2, each cheap to execute once in a transaction, which are costly to execute serially (*i.e.*, it is costly to execute R_1 and then R_2 in the same transaction). In this case, we can create a "combined" resource R_1R_2 which is itself metered separately. Note that, in our discussion of resources, there is no requirement that the resources be independent in any sense, and such "combined resources" are themselves very reasonable to consider.

Resource utilization targets. As mentioned previously, many networks have a minimum node requirement, implying a sustained target for resource utilization in each group of transactions added to the blockchain. (For simplicity, we will call this sequence of transactions a *block*, though it could be any collection of transactions that makes sense for a given blockchain.) We will write this resource utilization target as a vector $b^\star \in \mathbb{R}^m$ whose ith entry denotes the desired amount of resource i to be consumed in a block. The resource utilization of a particular block is a linear function of the transactions included in a block. We will denote the included transactions as a Boolean vector $x \in \{0,1\}^n$ whose jth entry is one if transaction j is included in the block and is zero otherwise. We will write $A \in \mathbb{R}^{m \times n}$ as a matrix whose jth column is the vector of resources a_j consumed by transaction j. We can then write the total quantity of consumed resources by this block as

$$y = Ax, \tag{1}$$

where $y \in \mathbb{R}^m$ is a vector whose ith entry denotes the quantity of resource i consumed by all transactions included in the block. Additionally, we can write the deviation from the target utilization, sometimes called the residual, as

$$Ax - b^\star,$$

i.e., a vector whose ith element is the total quantity of resource i, consumed by transactions in this block, minus the target $b_i^\star$ for resource i. For example, in Ethereum post EIP-1559, there is only one resource, gas, which has a target of 15M [29]. (We will see later how this notion of a resource utilization target can be generalized to a loss function.)

Resource utilization limits. In addition to resource targets, a blockchain may introduce a resource limit $b \in \mathbb{R}^m$ such that any valid block with transactions x must satisfy

$$Ax \leq b.$$

Continuing the example from before, Ethereum after EIP-1559 has a single resource, gas, with a resource limit of 30M.

Network fees. We want to incentivize users and miners to keep the total resource usage "close" to $b^\star$. To this end, we introduce a *network fee* $p_i \in \mathbb{R}$ for resource $i = 1, \dots, m$, which we will sometimes call the *resource price*, or just the *price*. If transaction j with resource vector a_j is included in a block, a fee $p^T a_j = \sum_i p_i (a_j)_i$ must be paid to the network. (In Ethereum, the network fee is implemented by burning some amount of the gas fee for each transaction and can be thought of as the "burn rate".) For now, we will assume that as the fee p_i gets larger, the amount of resource i used in a block $(Ax)_i$ will become smaller and vice versa.

Resource mispricing. Given A and b, it is, in general, not clear how to set the fees p in order to ensure the network has good performance (in other words, so that the resource utilization is "close" to $b^\star$). As a real world example, starting in Ethereum block number 2283416 (Sept. 18, 2016) an attacker exploited the fact that resources were mispriced for the EXTCODESIZE opcode, causing the network to slow down meaningfully. This mispricing was fixed via the hard fork on block 2463000 (Oct. 18, 2016) with details outlined in EIP-150 [12]. (The effects of this mispricing can still be observed when attempting to synchronize a full node. A dramatic slowdown in downloading and processing blocks happens starting at the previously mentioned block height.) Though usually less drastic, there have been similar events on other blockchains, underscoring the importance of correctly setting resource prices.

Setting fees. We want a simple update rule for the network fees p with the following properties:

1. If $Ax = b^\star$, then there is no update.
2. If $(Ax)_i > b^\star_i$, then p_i increases.
3. Otherwise, if $(Ax)_i < b^\star_i$, then p_i decreases.

There are many update rules with these properties. As a simple example, we can update the network fees as

$$p^{k+1} = \left(p^k + \eta(Ax - b^\star)\right)_+, \tag{2}$$

where $\eta > 0$ is some (usually small) positive number, often referred to as the "step size" or "learning rate", k is the block number such that p^k are the resource prices at block k, and $(w)_+ = \max\{0, w\}$ for scalar w and is applied elementwise for vectors. Recently, Ethereum developers [18] proposed the update rule

$$p^{k+1} = p^k \odot \exp\left(\eta(Ax - b^\star)\right). \tag{3}$$

Here, $\exp(\cdot)$ is understood to apply elementwise, while $\odot$ is the elementwise or Hadamard product between two vectors. The remainder of this paper will show that many update rules are attempting to (approximately) solve an instance of a particular optimization problem with a natural interpretation, where parts of the update rule come from a specific choice of objective function by the network designer. (We show in section 3.3 that a similar rule to (3) can be derived as a consequence of the framework presented in this paper, under a particular choice of variable transformation.)

We note that [31] has studied fixed points of such iterations in the one-dimensional case, extending the analysis of EIP-1559. However, the multidimensional scenario can be quite a bit more subtle to analyze. For instance, the multiplicative update rule (3) can admit "vanishing gradient" behavior in high-dimensions [34]. We suspect that the one-dimensional fee model of [31] can be extended to the multidimensional rules (2) and (3) but leave this for future work.

3 Resource allocation problem

As system designers, our ultimate goal is to maximize utility of the underlying blockchain network by appropriately allocating resources to transactions. However, we cannot perform this allocation directly, since we cannot control what users and miners wish to include in blocks, nor do we know what the utility of a transaction is to users and miners. Instead, we aim to set the fees p to ensure that the resource usage is approximately equal to the desired target, which we will represent as an objective function. We will show later that the update mechanisms proposed above naturally fall out of a more general optimization formulation. Similarly to the Transmission Control Protocol (TCP), each update rule is a result of a particular objective function, chosen by the network designer [42, 41].

Loss function. We define a *loss function* of the resource usage, $\ell : \mathbb{R}^m \to \mathbb{R} \cup \{\infty\}$, which maps a block's resource utilization, y, to the "unhappiness" of the network designer, $\ell(y)$. We assume only that ℓ is convex and lower semicontinuous. (We will not require monotonicity, nonnegativity, or other assumptions on ℓ, but we will show that useful properties do hold in these scenarios.)

For example, the loss function can encode "infinite dissatisfaction" if the resource target is violated at all:

$$\ell(y) = \begin{cases} 0 & y = b^\star \\ \infty & \text{otherwise.} \end{cases} \tag{4}$$

(Functions of this form, which are either 0 or ∞ at every point, are known as *indicator functions.*) Note also that this loss is not differentiable anywhere, but it is convex. Another possible loss, which is also an indicator function, is

$$\ell(y) = \begin{cases} 0 & y \le b^\star \\ \infty & \text{otherwise.} \end{cases} \tag{5}$$

This loss can roughly be interpreted as: we do not mind any usage below $b^\star$, but we are infinitely unhappy with any usage above the target amounts. Alternatively, we may only care about large deviations from the target $b^\star$:

$$\ell(y) = (1/2)\|y - b^\star\|_2^2,$$

or, perhaps, require that the loss is simply linear and independent of $b^\star$,

$$\ell(y) = u^T y, \tag{6}$$

for some fixed vector $u \in \mathbb{R}^m$. Another important family of losses are those which are separable and depend only on the individual resource utilizations,

$$\ell(y) = \sum_{i=1}^{m} \phi_i(y_i) \tag{7}$$

where $\phi_i : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ for $i = 1, \dots, m$, are convex, nondecreasing functions. (The loss (5) is a special case of this loss, while (6) is a special case when the vector u is nonnegative.) We will make the technical assumption that $\phi_i(0) < \infty$ for every i, otherwise no resource allocation would have finite loss.

We will see that each definition of a loss function implies a particular update rule for the network fees p. This utility function can more generally be engineered to appropriately capture tradeoffs in increasing throughput of a particular resource at the possible detriment of other resources.

Resource constraints. Now that we have defined the network designer's loss, which is a way of quantifying "unhappiness" when the resource usage is y, we need some way to define the transactions that users are willing to create and, importantly, that miners are willing (and able) to include. We do this in a very general way by letting $S \subseteq \{0,1\}^n$ be the set of possible transactions that users and miners are willing and able to create and include. Note that this set is discrete and can be very complicated or potentially hard to maximize over (as is the case in practice). For example, the set S could encode a demand for transactions which depend on other transactions being included in the block (as is the case in, *e.g.*, miner extractable value [37, 26, 46]), network-defined hard constraints of certain resources (such as $Ax \le b$ for every $x \in S$), and even very complicated interactions among different transactions (if certain contracts can, for example, only be called a fixed number of times, as in NFT mints). We make no assumptions about the structure of this set S but only require that the included transactions, $x \in \{0,1\}^n$, obey the constraint $x \in S$.

Convex hull of resource constraints. A network designer may be more interested in the long-term resource utilization of the network than the resource utilization of any one particular block. In this case, the designer may choose to "average out" each transaction over a number of blocks instead of deciding whether or not to include it in the next block. To that end, we, as designers, will be allowed to choose convex combinations of elements of the constraint set S, which we will write as $\mathbf{conv}(S)$. (In general, this means that we can pick probability distributions over the elements of S, and x is allowed to be the expectation of any such probability distribution; *i.e.*, we only require that, for the designer, x is reasonable "in expectation".) Here, components of x may vary continuously between 0 and 1; these values have a simple interpretation. If x_i is not 0 or 1, then we can interpret the quantity $1/x_i$ as roughly the number of blocks after which transaction i is included. Of course, neither users nor miners can choose transactions to be "partially included", so this property will only apply to the idealized problem we present below. While this relaxation might seem unrealistically "loose", we will see later how this easily translates to the realistic case where transactions are either included or not (that is, x_i is either 0 or 1) by users and miners.

Transaction utility. Finally, we introduce the *transaction utilities*, which we will write as $q \in \mathbb{R}^n$. The transaction utility q_j for transaction $j = 1, \dots, n$ denotes the users' and miners' joint utility of including transaction j in a given block. Note that it is very rare (if at all possible) to know the values of q. However, we will see that, under mild assumptions, we do not need to know the values of q in order to get reasonable prices, and reasonable update rules will not depend on q.

Resource allocation problem. We are now ready to write the *resource allocation problem*, which is to maximize the utility of the included transactions, minus the loss, over the convex hull of possible transactions:

$$\begin{array}{ll} \text{maximize} & q^T x - \ell(y) \\ \text{subject to} & y = Ax \\ & x \in \mathbf{conv}(S). \end{array} \tag{8}$$

This problem has variables $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, and the problem data are the resource matrix $A \in \mathbb{R}^{m \times n}$, the set of possible transactions $S \subseteq \{0,1\}^n$, and the transaction utilities $q \in \mathbb{R}^n$. Because the objective function is concave and the constraints are all convex, this is a convex optimization problem. On the other hand, even though the set $\mathbf{conv}(S)$ is convex, it is possible that $\mathbf{conv}(S)$ does not admit an efficient representation (for example, it may contain exponentially many constraints) which means that solving this problem is, in general, not easy.

Interpretation. We can interpret the resource allocation problem (8) as the "best case scenario", where the designer is able to choose which transactions are included (or "partially included") in a block in order to maximize the total utility. While this problem is not terribly useful by itself, since (a) it cannot really be implemented in practice, (b) we often don't know q, and (c) we cannot "partially include" a transaction within a block, we will see that it will decompose naturally into two problems. One of these problems can be easily solved on chain, while the other is solved implicitly by the users (who send transactions to be included) and miners (who choose which transactions to include). The solutions to the latter problem can always be assumed to be integral; *i.e.*, no transactions are "partially included". This will allow us to construct a simple update rule for the prices, which does not depend on q. For the remainder of the paper, we will call this combination of users and miners the *transaction producers*.

Offchain agreements and producers. Due to the inevitability of user-miner collusion, we consider the combination of the two, the transaction producers, as the natural unit. For example, it is not easily possible to create a transaction mechanism where the users are forced to pay miners some fixed amount, since it is always possible for miners to refund users via some off-chain agreement [48]. Similarly, we cannot force miners to accept certain transactions from users, since a miner always has plausible deniability of not having seen a given transaction in the mempool. While not perfect for a general analysis of incentives, this conglomerate captures the dynamics between the network's incentives and those of the miners and users better than assuming each is purely selfishly maximizing their own utility (as opposed to strategically colluding) and suffices for our purposes.

3.1 Setting prices using duality

In this section, we will show a decomposition method for this problem. This decomposition method suggests an algorithm (presented later) for iteratively updating fees in order to maximize the transaction producers' utility minus the loss of the network, given historical observations.

To start, we will reformulate (8) slightly by pulling the constraint $x \in \mathbf{conv}(S)$ into the objective,

$$\begin{array}{ll} \text{maximize} & q^T x - \ell(y) - I(x) \\ \text{subject to} & y = Ax \end{array} \tag{9}$$

where $I : \mathbb{R}^n \to \mathbb{R} \cup \{\infty\}$ is the indicator function defined as $I(x) = 0$ if $x \in \mathbf{conv}(S)$ and $I(x) = +\infty$ otherwise.

Dual function. The Lagrangian [11, §5.1.1] for problem (9) is then

$$L(x, y, p) = q^T x - \ell(y) - I(x) + p^T (y - Ax),$$

with dual variable $p \in \mathbb{R}^m$. This corresponds to "relaxing" the constraint $y = Ax$ to a penalty $p^T(y - Ax)$ assigned to the objective, where the price per unit violation of constraint i is p_i. (Negative values denote refunds.) Rearranging slightly, we can write

$$L(x, y, p) = p^T y - \ell(y) + (q - A^T p)^T x - I(x).$$

The corresponding dual function [11, §5.1.2], which we will write as $g : \mathbb{R}^m \to \mathbb{R} \cup \{-\infty\}$, is found by partially maximizing L over x and y:

$$g(p) = \sup_y \left(p^T y - \ell(y)\right) + \sup_x \left((q - A^T p)^T x - I(x)\right). \tag{10}$$

Discussion. The first term can be recognized as the Fenchel conjugate of ℓ [11, §3.3] evaluated at p, which we will write as $\ell^*(p)$, while the second term is the optimal value of the following problem:

$$\begin{array}{ll} \text{maximize} & (q - A^T p)^T x \\ \text{subject to} & x \in \mathbf{conv}(S), \end{array} \tag{11}$$

with variable $x \in \mathbb{R}^n$. We can interpret this problem as the transaction producers' problem of creating and choosing transactions to be included in a block in order to maximize their utility, after netting the fees paid to the network. (We will sometimes refer to this problem as the transaction producers' *packing problem.*) We note that the optimal value of (11) in terms of p, which we will write as $f(p)$, is the pointwise supremum of a family of linear (and therefore convex) functions of p, so it, too, is a convex function [11, §3.2.3]. Note that since the objective is linear, problem (11) has the same optimal objective value as the nonconvex problem

$$\begin{array}{ll} \text{maximize} & (q - A^T p)^T x \\ \text{subject to} & x \in S, \end{array}$$

where we have replaced $\mathbf{conv}(S)$ with S. (See full version [27] appendix A.2 for a simple proof.) Finally, since $f(p)$ is the optimal value of problem (11) for fees p, the dual function g can be written as

$$g(p) = \underbrace{\ell^*(p)}_{\text{network}} + \underbrace{f(p)}_{\text{tx producers}}.$$

Since the dual function g is the sum of convex functions ℓ^* and f, it, too, is convex. (We will make use of this property soon.) Having defined the dual function g, we will see how this function can give us a criterion for how to best set the network fees p.

Duality. An important consequence of the definition of the dual function g is *weak duality* [11, §5.2.2]. Specifically, letting $s^\star$ be the optimal objective value for problem (8), we have that $g(p) \geq s^\star$ for every possible choice of price $p \in \mathbb{R}^m$. This is true because we have essentially "relaxed" the constraint to a penalty, so any feasible point x, y for the original problem (9) always has 0 penalty. (There may, of course, be other points that are not feasible for (9) but are perfectly reasonable for this "relaxed" version, so we've only made the set of possibilities larger.) The proof is a single line:

$$g(p) = \sup_{x,y} L(x,y,p) \geq \sup_{y=Ax} L(x,y,p) = \sup_{y=Ax} \big(q^T x - \ell(y) - I(x)\big) = s^\star.$$

A deep and important result in convex optimization is that, in fact, there exists a $p^\star$ for which

$$g(p^\star) = s^\star,$$

under some basic constraint qualifications.[1] In other words, adding the constraint $y = Ax$ to the problem is identical to correctly setting the prices p. Since we know for any p that $g(p) \geq s^\star$ then

$$g(p^\star) = \inf_p g(p),$$

or, that $p^\star$ is a minimizer of g. This motivates an optimization problem for finding the prices.

[1] The condition is that the relative interior of $A\,\mathbf{conv}(S) \cap \mathbf{dom}\,\ell$ is nonempty. Here, we write $A\,\mathbf{conv}(S) = \{Ax \mid x \in \mathbf{conv}(S)\}$ and $\mathbf{dom}\,\ell = \{x \mid \ell(x) < \infty\}$, while the relative interior is taken with respect to the affine hull of the set. This condition almost always holds in practice for reasonable functions ℓ and sets S.

The dual problem. The *dual problem* is to minimize g, as a function of the fees p. In other words, the dual problem is to find the optimal value of

$$\text{minimize} \quad g(p), \tag{12}$$

with variable $p \in \mathbb{R}^m$. If we can easily evaluate g, then, since this problem is a convex optimization problem, as g is convex, solving it is usually also easy. An optimizer of the dual problem has a simple interpretation using its optimality conditions. Let $p^\star$ be a solution to the dual problem (12) for what follows. If the packing problem (11) has a unique solution $x^\star$ for $p^\star$, then the objective value f is differentiable at $p^\star$. (See full version [27] appendix A.2.) Similarly, under mild conditions on the loss function ℓ (such as strict convexity) the function ℓ^* is differentiable at $y^\star$, with derivative satisfying $\nabla \ell(y^\star) = p^\star$. In this case, the optimality conditions for problem (12) are that

$$\nabla g(p^\star) = y^\star - Ax^\star = 0. \tag{13}$$

In other words, the fees $p^\star$ that minimize (12) are those that charge the transaction producers the exact marginal costs faced by the network, $\nabla \ell(Ax^\star) = p^\star$. Furthermore, these are exactly the fees which incentivize transaction producers to include transactions that maximize the welfare generated minus the network loss, subject to resource constraints, since $y^\star$ and $x^\star$ are feasible and optimal for problem (8).

Differentiability. In general, g is not always differentiable, but is almost universally subdifferentiable, under mild additional conditions on ℓ (*e.g.*, ℓ does not contain a line). Condition (13) may then be replaced with

$$0 \in \partial g(p^\star) = -Y^\star(p^\star) + AX^\star(p^\star),$$

where

$$Y^\star(p) = \operatorname*{argmax}_y \left(p^T y - \ell(y)\right),$$

while $X^\star(p) \subseteq \mathbf{conv}(S)$ are the maximizers of problem (11) for price p. We define $AX^\star(p) = \{Ax \mid x \in X^\star(p)\}$, and write $\partial g(p^\star)$ for the subgradients of g at $p^\star$ (*cf.*, full version [27] appendix A). The condition then says that the intersection of the extremizing sets $Y^\star(p^\star)$ and $AX^\star(p^\star)$ is nonempty at the optimal prices $p^\star$. We show a special case of this below, when $p = 0$, with a direct proof using strong duality that does not require these additional conditions.

3.2 Properties

There are a number of properties of the prices p that can be derived from the dual problem (12).

Nonnegative prices. If the objective function ℓ is separable and nondecreasing, as in (7), then any price p_i feasible for problem (12) must be nonnegative, $p_i \geq 0$. (By feasible, we mean that $g(p) < \infty$.) To see this, note that, by definition (7), we have

$$\ell^*(p) = \sup_y \left(p^T y - \ell(y)\right) = \sum_{i=1}^m \sup_{y_i} \left(p_i y_i - \phi_i(y_i)\right),$$

so we can consider each term individually. If $p_i < 0$ then any $y_i \leq 0$ must have

$$p_i y_i - \phi_i(y_i) \geq p_i y_i - \phi_i(0) \to \infty,$$

as $y_i \to -\infty$ since $\phi_i(y_i)$ is nondecreasing in y_i. So $g(p) \to \infty$ and therefore this choice of p cannot be feasible, so we must have that $p_i \geq 0$.

Superlinear separable losses. If the losses ϕ_i are superlinear, in that

$$\frac{\phi_i(z)}{z} \to \infty, \tag{14}$$

as $z \to \infty$ and bounded from below, in addition to being nondecreasing, then $\ell^*(p)$ is finite for $p \geq 0$. This means that the effective domain of g, defined as the set of prices for which g is finite,

$$\mathbf{dom}\, g = \{p \in \mathbb{R}^m \mid g(p) < \infty\},$$

is exactly the nonnegative orthant. (This discussion may appear somewhat theoretical, but we will see later that it turns out to be an important practical point when updating prices.) While not all losses are superlinear, we can always make them so by, *e.g.*, adding a small, nonnegative squared term to ϕ_i, say

$$\tilde{\phi}_i(z) = \phi_i(z) + \rho(z)_+^2,$$

where $(z)_+ = \max\{0, z\}$ and $\rho > 0$, or by setting $\phi_i(z) = \infty$ for $z \geq 0$ large enough.

Subsidies. Alternatively, if the function ℓ is decreasing somewhere on the interior of its domain, then there exist points $y^\star$ for which prices p_i are negative – *i.e.*, sometimes the network is willing to subsidize usage by paying users to use the network to meet its intended target. The interpretation is simple: if the base demand of the network is not enough to meet the target usage, then the network has an incentive to subsidize users until the marginal cost of the target usage matches the subsidy amounts. We note that this may only apply to very specific transaction types in practice, as it is difficult to issue subsidies in an incentive-compatible manner that doesn't encourage the inclusion of "junk" transactions.

Maximum price. There often exist prices past which transaction producers would always prefer to not submit a transaction (or, more generally, will only submit transactions that consume no resources, if such transactions exist). In fact, we can characterize the set of all such prices.

To do this, write $S_0 \subseteq S$ for the set of transactions bundles that use no resources, defined

$$S_0 = \{x \in S \mid Ax = 0\}.$$

If $0 \in S$ then S_0 is nonempty (as $0 \in S_0$), and we usually expect this set to be a singleton, $S_0 = \{0\}$. Otherwise, we are saying that there are transactions that are always costless to include. Now, we will define the set

$$P = \bigcap_{x \in S \setminus S_0} \{p \in \mathbb{R}_+^m \mid p^T Ax > q^T x\}.$$

This is the set of prices $p \in P$ such that, for every possible transaction bundle $x \in S$, the price of this transaction bundle, $p^T Ax$, paid to the network, is strictly larger than the total welfare generated by including it, which is $q^T x$. (That is, any transaction bundle x that is not costless is always strictly worse than no transaction, for transaction producers, at these prices.) The set P is nonempty since $\mathbf{1}^T Ax > 0$ for every $x \in S \setminus S_0$ (and $S \setminus S_0$ is finite) so, setting $p = t\mathbf{1}$, we have that

$$p^T Ax - q^T x = t\mathbf{1}^T Ax - q^T x \to \infty > 0,$$

as $t \to \infty$, so $t\mathbf{1} \in P$ for t large enough. The set P is also a convex set, as it is the intersection of convex sets. Additionally, if $p \in P$, then any prices p' satisfying $p' \geq p$ must also have $p' \in P$, where the inequality is taken elementwise. In English: if certain resource prices $p \in P$ would mean that transactions that consume resources are not included, then increasing the price of any resources to $p' \geq p$ also implies the same.

3.3 Solution methods

As mentioned before, the dual problem (12) is convex. This means that it can often be easily solved if the function g (or its subgradients) can be efficiently evaluated. We will see that, assuming users and miners are approximately solving problem (11), we can retrieve approximate (sub)gradients of g and use these to (approximately) solve the dual problem (12). In a less-constrained computational environment, a quasi-Newton method (*e.g.*, L-BFGS) would converge quickly to the optimal prices and be efficient to implement. However, these methods aren't amenable to on-chain computation due to their memory and computational requirements. To solve for the optimal fees on chain, we therefore propose a modified version of gradient descent which is easy to compute and does not require additional storage beyond the fees themselves.

Projected gradient descent. A common algorithm for unconstrained function minimization problems, such as problem (12), is *gradient descent.* In gradient descent, we are given an initial starting point p^0 and, if the function g is differentiable, we iteratively update the prices in the following way:

$$p^{k+1} = p^k - \eta \nabla g(p^k).$$

Here, $\eta > 0$ is some (usually small) positive number referred to as the "step size" or "learning rate" and $k = 0, 1, \dots$ is the iteration number. This rule has a few important properties. For example, if $\nabla g(p^k) = 0$, that is, p^k is optimal, then this rule does not update the prices, $p^{k+1} = p^k$; in other words, any minimizer of g is a fixed point of this update rule. Additionally, this rule can be shown to converge to the optimal value under some mild conditions on g, *cf.* [9, §1.2]. This update also has a simple interpretation: if $\nabla g(p^k)$ is not zero, then a small enough step in the direction of $\nabla g(p^k)$ is guaranteed to evaluate to a lower value than p^k, so an update in this direction decreases the objective g. (This is why the parameter η is usually chosen to be small.)

Note that if the effective domain of the function g, $\mathbf{dom}\, g$, is not $\mathbb{R}^m$, then it is possible that the $(k+1)$st step ends up outside of the effective domain, $p^{k+1} \notin \mathbf{dom}\, g$, so $g(p^k) = \infty$ which would mean that the gradient of g at price p^{k+1} would not exist. To avoid this, we can instead run *projected* gradient descent, where we project the update step into the domain of g, in order to get $p^{k+1} \in \mathbf{dom}\, g$, *i.e.*,

$$p^{k+1} = \mathbf{proj}(p^k - \eta \nabla g(p)) \tag{15}$$

where $\mathbf{proj}(z)$ is defined

$$\mathbf{proj}(z) = \operatorname*{argmin}_{p \in \mathbf{dom}\, g} \|z - p\|_2^2.$$

In English, $\mathbf{proj}(z)$ is the projection of the price to the nearest point in the domain of g, as measured by the sum-of-squares loss $\|\cdot\|_2^2$. (This always exists and is unique as the domain of g is always closed and convex, for any loss function ℓ as defined above.) There is relatively rich literature on the convergence of projected gradient descent, and we refer the reader to [50, 9] for more.

Evaluating the gradient. In general, since we do not know q, we cannot evaluate the function g at a certain point, say p^k. On the other hand, the gradient of g at p^k, when g is differentiable, depends only on the solution to problem (11) and the maximizer for the conjugate function ℓ^*, at the price p^k. (This follows from the gradient equation in (13).) So, if we know that transaction producers are solving their welfare maximization problem (11) to (approximate) optimality, equation (13) suggests a simple descent algorithm for solving the dual problem (12).

From before, let $y^\star$ be a maximizer of $\sup_y \left(y^T p^k - \ell(y)\right)$, which is usually easy to compute in practice, and let x^0 be an (approximate) solution to the transaction inclusion problem (11) (observed, *e.g.*, after the block is built with resource prices p^k). We can approximate the gradient of g at the current fees p using (13), where we replace the true solution $x^\star$ with the observed solution x^0. Since x^0 is Boolean, we can compute the resource usage Ax^0 after observing only the included transactions. We can then update the fees p^k in, say, block k, to a new value p^{k+1} by using projected gradient descent with this new approximation:

$$p^{k+1} = \mathbf{proj}(p^k - \eta(y^\star - Ax^0)). \tag{16}$$

Discussion. Whenever ℓ is differentiable at $y^\star$, we have that $\nabla\ell(y^\star) = p$. (To see this, apply the first-order optimality conditions to the objective in the supremum in the definition of $\ell^\star$.) We can then think of $y^\star$ as the resource utilization such that the marginal cost to the network $\nabla\ell$ is equal to the current fees p. Thus, the network aims to set p such that the realized resource utilization is equal to $y^\star$. We can see that (15) will increase the network fee for a resource being overutilized and decrease the network fee for a resource being underutilized. This pricing mechanism updates fees to disincentivize future users and miners from including transactions that consume currently-overutilized resources in future blocks. Additionally, we note that algorithms of this form are not the only algorithms which are reasonable. For example, any algorithm that has a fixed point p satisfying $\nabla g(p) = 0$ and converges to this point under suitable conditions is also similarly reasonable. One well-known example is an update rule of the form of (3):

$$p^{k+1} = p^k \odot \exp(-\eta \nabla g(p^k)),$$

when the prices must be nonnegative, *i.e.*, when $\mathbf{dom}\, g \subseteq \mathbb{R}^m_+$. We note that one important part of reasonable rules is that they only depend on (an approximation of) the gradient of the function g, since the value of g may not even be known in practice. Additionally, in some cases, the function g is nondifferentiable at prices p. In this case, the subgradient still often exists and convergence of the update rule (16) can be guaranteed under slightly stronger conditions. (The modification is needed as not all subgradients are descent directions.)

Simple examples. We can derive specific update rules by choosing specific loss functions. For example, consider the loss function

$$\ell(y) = \begin{cases} 0 & y = b^\star \\ +\infty & \text{otherwise}, \end{cases}$$

which captures infinite unhappiness of the network designer for any deviation from the target resource usage $b^\star$. The corresponding conjugate function is

$$\ell^*(p) = \sup_y (y^T p - \ell(y)) = (b^\star)^T p,$$

with maximizer $y^\star = b^\star$. (Note that this maximizer does not change for any price p). Since $\mathbf{dom}\, g = \mathbb{R}^m$, then the update rule is

$$p^{k+1} = p^k - \eta(b^\star - Ax^0). \tag{17}$$

If the utilization Ax^0 lies far below $b^\star$, the fees p^k might become negative, *i.e.*, the network would subsidize certain resource usage to meet the requirement that it must be equal to $b^\star$.

Nondecreasing separable loss. A more reasonable family of loss functions would be the nondecreasing, separable losses:

$$\ell(y) = \sum_{i=1}^{m} \phi_i(y_i).$$

From §3.2 we know that the domain of g is precisely the nonnegative orthant when the functions ϕ_i are superlinear (*i.e.*, satisfy (14)) and bounded from below, so we have that $\mathbf{proj}(z) = (z)_+$, where $(w)_+ = \max\{0, w\}$ for scalar w and is applied elementwise for vectors. Additionally, using the definition of the separable loss, we can write

$$\ell^*(p) = \sum_{i=1}^{m} \sup_{y_i} \left(y_i p_i - \phi_i(y_i)\right).$$

Letting $y_i^\star$ be the maximizers for the individual problems at the current price p^k, we have

$$p^{k+1} = (p^k - \eta(y^\star - Ax^0))_+.$$

For example, if ϕ_i is an indicator function with $\phi_i(y_i) = 0$ if $y_i \le b_i^\star$ and ∞ otherwise, as in the loss (5), then an optimal point is always $y_i^\star = b_i^\star$, when $p_i \ge 0$. Since no updates will ever set $p_i < 0$, we therefore have,

$$p^{k+1} = (p^k - \eta(b^\star - Ax^0))_+,$$

which is precisely the update given in (2).

Exponential update rule. A log transform of the prices leads to an update rule resembling (3), proposed by Ethereum developers [18]. We assume that the loss function is separable and nondecreasing (*cf.*, §3.2) and that a minimum demand condition is met (given in the full version [27, §3.2]) so that the prices are strictly positive. As a result, we can write the prices p as $p = \exp(\tilde{p})$ for some $\tilde{p} \in \mathbb{R}^m$, where the exponential is applied elementwise. The gradient with respect to $\tilde{p}$ is $\nabla g(\tilde{p}) = \nabla g(p) \odot \exp(\tilde{p})$. Writing the resulting gradient update for $\tilde{p}$ and then taking the exponential of both sides, we get

$$p^{k+1} = p^k \odot \exp\left(-p^k \odot \eta \nabla g(p^k)\right).$$

When we use the indicator loss function (4), we obtain a rule similar to the original EIP-1559 update (3) but with an extra factor of p^k in the exponent:

$$p^{k+1} = p^k \odot \exp(\eta(p^k \odot (Ax^k - b^\star))). \tag{18}$$

The lack of this extra factor in EIP-1559 may explain some of the behavior explored in [40].

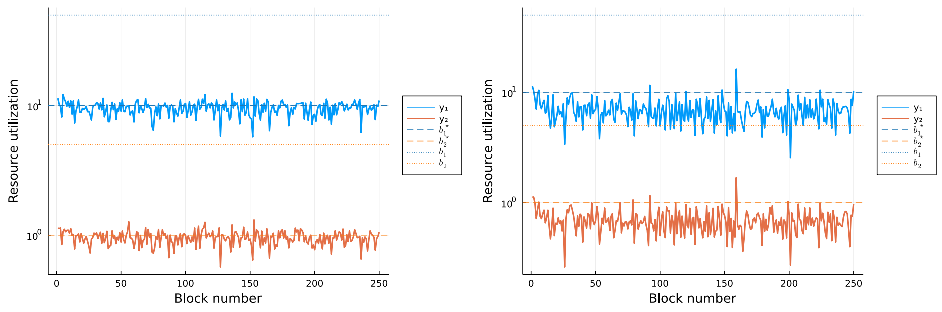

Figure 1 Resource utilization for multidimensional pricing (left) clusters closer to target values than for uniform pricing (right), which includes limited information about individual targets or caps.

4 The cost of uniform prices

In this section, we show that pricing resources using the method outlined above can increase network efficiency and make the network more robust to DoS attacks and resource demand distribution shifts. We construct a toy experiment to illustrate these differences between uniform and multidimensional resource pricing but leave more extensive numerical studies to future work.

The setup. We consider a blockchain system with two resources (*e.g.*, compute and storage) with resource utilizations y_1 and y_2. Resource 1 is much cheaper for the network to use than resource 2, so $b^\star = (10, 1)$ and $y \leq b = (50, 5)$. Furthermore, we assume that there is a joint capacity constraint on these resources, $y_1 + 10y_2 \leq 50$, which captures the resource tradeoff. Each transaction a_j is therefore a vector in $\mathbb{R}^3_+$ with $a_j = (a_{1j}, a_{2j}, a_{1j} + 10a_{2j})$. As in §3.3, we consider the simple loss function $\ell(y) = 0$ if $y = b^\star$ and $+\infty$ otherwise, which has the update rule given in (17). In the scenarios below, we compare our multidimensional fee market approach to a baseline, where both resources are combined into one equal to $a_{1j} + 10a_{2j}$ with $b^\star = 20\% \times \max(b_1, b_2) = 10$. We demonstrate that pricing these resources separately leads to better network performance. All code is available at

`https://github.com/bcc-research/resource-pricing`.

We run simulations in the Julia programming language [10]. The transaction producers' optimization problem (11) is modeled with JuMP [28] and solved with COIN-OR's simplex-based linear programming solver, Clp [32]. The solution is usually integral, but when it is not, we fall back to the HiGHS mixed-integer linear program solver [35].

Scenario 1: steady state behavior. We consider a sequence of 250 blocks. At each block, there are 15 submitted transactions, with resource usage randomly drawn as $a_{1j} \sim \mathcal{U}(0.5, 1)$ and $a_{2j} \sim \mathcal{U}(0.05, 0.1)$. (For example, these may be moderate compute and low storage transactions.) Transaction utility is drawn as $q_j \sim \mathcal{U}(0, 5)$. We initialize the price vector as $p = 0$ and examine the steady state behavior, where the price updates and transaction producer behavior are defined as in the previous section. We use a learning rate $\eta = 1 \times 10^{-2}$ throughout. The resource utilization, shown in figure 1, suggests that our multidimensional scheme more closely tracks the target utilities $b^\star$ than a single-dimensional fee market. Furthermore, the number of transactions included per block is consistently higher, illustrated in figure 2 (purple line).

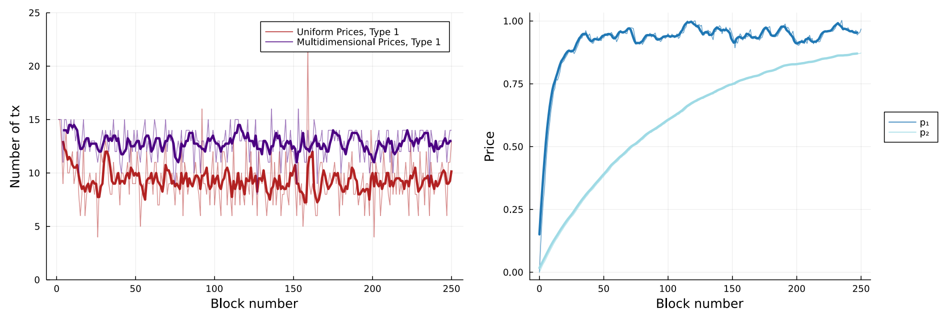

Figure 2 Multidimensional pricing allows us to include more transactions per block (left) by optimally adjusting prices (right). The thicker line is the four-sample moving average of the data.

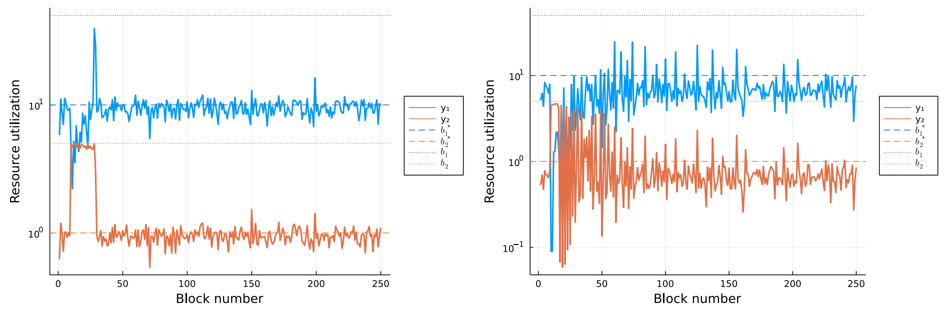

Figure 3 Resource utilization for multidimensional pricing (left) clusters closer to target values than for uniform pricing (right) after a burst to the resource limit to handle transactions that make heavy use of resource 2.

Scenario 2: transactions distribution shift. Often, the distribution of transaction types submitted to a blockchain network differs for a short period of time (*e.g.*, during NFT mints). There may be a change in both the number of transactions and the distribution of resources required. We repeat the above simulation but add 150 transactions in block 10 with resource vector $a_j = (0.01, 0.5)$. (For example, these transactions may have low computation but high storage requirements.) We draw the utility $q_j \sim \mathcal{U}(10, 20)$ and begin the network at the steady-state prices from scenario 1. In figure 3, we see that a multidimensional fee market gracefully handles the distribution shift. The network fully utilizes resource 2 for a short period of time before returning to steady state. Uniform pricing, on the other hand, does not do a good job of adjusting its resource usage and oscillates around the target. Figure 4 show that, as a result, multidimensional pricing is able to include more transactions, both during the distribution shift and after the network returns to steady state. We see that the prices smoothly adjust accordingly.

5 Extensions

Parallel transaction execution model. Consider the scenario where the nodes have L parallel execution environments (*e.g.*, threads), each of which has its own set of m identical resources. In addition, there are r resources shared between the environments. We denote transactions run on thread k by $x_k \in \{0, 1\}^n$. The resource allocation problem becomes

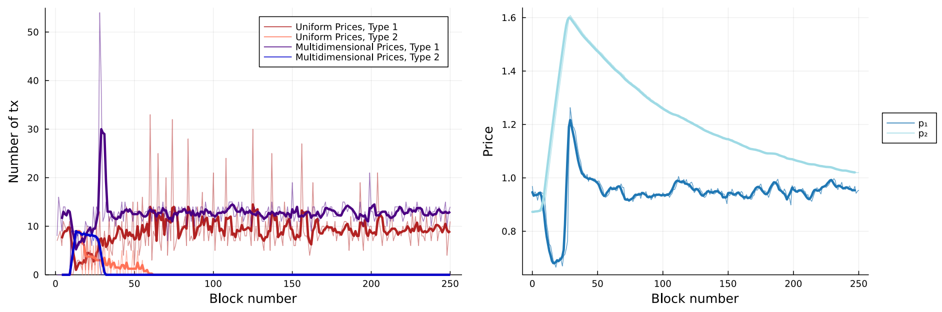

Figure 4 Multidimensional pricing allows us to include more transactions per block by optimally adjusting prices (right). The thicker line is the four-sample moving average of the data.

$$\begin{array}{ll} \text{maximize} & \sum_{k=1}^{L} q^T x_k - \ell(y_1, \dots, y_L, y^{\text{shared}}) \\ \text{subject to} & y_k = Ax_k, \quad x_k \in \mathbf{conv}(S) \qquad k = 1, \dots, L \\ & y^{\text{shared}} = Bz, \quad z = \sum_{k=1}^{L} x_k, \quad z \in \mathbf{conv}(S^{\text{shared}}) \end{array}$$

As before, the Boolean vector sets S and S^{shared} encode constraints such as resource limits for each parallel environment and the shared environment respectively. In addition, we'd expect to have $z \leq \mathbf{1}$ if each transaction is only allocated to a single environment, which can be encoded by S^{shared}. By stacking the variables into one vector, this problem can be seen as a special case of (8) and can be solved with the same method presented in this work. (The interpretation here is that we are declaring a number of "combined resources", each corresponding to the parallel execution environments along with their shared resources.)

Different price update speeds. Some resources may be able to sustain burst capacities for much shorter periods of times than other resources. In practice, we may wish to increase the prices of these resources faster. For example, a storage opcode that generates a lot of memory allocations will quickly cause garbage collection overhead, which could slow down the network. We can update (15) to include a learning rate for each resource. These learning rates can be chosen by system designers using simulations and historical data. We collect these in a diagonal matrix $D = \mathbf{diag}\,(\eta_1, \dots, \eta_m)$ and change the update rule to be

$$p^{k+1} = \mathbf{proj}(p^k - D\nabla g(p^k)).$$

Contract throughput. Alternatively, we can define utilization on a per-contract basis instead of a per-resource basis (per-contract fees were recently proposed by the developers of Solana [54]). We define the utilization of a smart contract j as $z_j = (w^T a_j) x_j$, where w is some weight vector and $x_j \in \mathbb{Z}_+$ is the number of times contract j is called. In matrix form, $z = A^T w \odot x$, where $\odot$ is the Hadamard (elementwise) product. For each contract, the utilization z_j is 0 when $x_j = 0$, which can be interpreted as not calling contract j in a block. When $x_j > 0$, the utilization is $\left(\sum_i w_i a_{ij}\right) x_j$. When we use per-contract utilizations, the loss function can capture a notion of fairness in resource allocation to contracts. For example, we may want to prioritize cheaper-to-execute contracts over more expensive ones by using, *e.g.*, proportional fairness as in [36], though there are many other notions that may be useful. With this setup, the resource allocation problem is

$$\begin{array}{ll} \text{maximize} & q^T x - \ell(z) \\ \text{subject to} & z = A^T w \odot x \\ & x \in \mathbf{conv}(S). \end{array}$$

Again, we can introduce the dual variable $p \in \mathbb{R}^n$ for the equality constraint, and, with a similar method to the one introduced in this paper, iteratively update this variable to find the optimal fees to charge for each smart contract call.

6 Conclusion

We constructed a framework for multidimensional resource pricing in blockchains. Using this framework, we modeled the network designer's goal of maximizing transaction producer utility, minus the loss incurred by the network, as a an optimization problem. We used tools from convex optimization – and, in particular, duality theory – to decompose this problem into two simpler problems: one solved on chain by the network, and another solved off chain by the transaction producers. The prices that unify the competing objectives of minimizing network loss and maximizing transaction producer utility are precisely the dual variables in the optimization problem. Setting these prices correctly (*i.e.*, to minimize the dual function) results in a solution to the original problem. We then demonstrated efficient methods for updating prices that are amenable to on-chain computation and derived an EIP-1559-like mechanism as an example. In a simple numerical example, we find that the proposed mechanism allows the network to equilibrate to its resource utilization target more quickly than uniform pricing, while offering greater throughput without increasing node hardware requirements. Finally, we show a number of simple extensions to our framework that capture other proposed mechanisms such as per-contract fees. We find that it allows the network to equilibrate to its resource utilization target more quickly than the uniform price case and offers greater throughput without increasing node requirements.

To the best of the authors' knowledge, this is the first work to systematically study optimal pricing of resources in blockchains in the many-asset setting. Future work and improvements to this model include a detailed game-theoretic analysis, extending that of [31], along with a more concrete analysis of the dynamical behavior of fees set in this manner. Finally, a thorough numerical evaluation of these methods under realistic conditions (such as testnets) will be necessary to see if these methods are feasible in production.

References

1 John Adler. Eip-2242: Transaction postdata, 2019. URL: `https://eips.ethereum.org/EIPS/eip-2242`.

2 John Adler. Multi-threaded data availability on eth 1. Ethresearch, 2019. URL: `https://ethresear.ch/t/multi-threaded-data-availability-on-eth-1/5899`.

3 John Adler. Accounts, strict access lists, and UTXOs - research / execution, 2020. URL: `https://forum.celestia.org/t/accounts-strict-access-lists-and-utxos/37`.

4 John Adler. Wait, it's all resource pricing? EthCC, 2021. URL: `https://www.youtube.com/watch?v=YoWMLoeQGeI`.

5 John Adler. Always has been (or, wait, it's all resource pricing? part 2). EthCC, 2022. URL: `https://www.youtube.com/watch?v=Zq8uwpX39oI`.

6 Akshay Agrawal, Stephen Boyd, Deepak Narayanan, Fiodar Kazhamiaka, and Matei Zaharia. Allocation of fungible resources via a fast, scalable price discovery method. *Mathematical Programming Computation*, pages 1–30, 2022.

7 Mustafa Al-Bassam. LazyLedger: A distributed data availability ledger with client-side smart contracts. *CoRR*, abs/1905.09274, 2019. `arXiv:1905.09274`.

8 Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities, 2018. doi:10.48550/ARXIV.1809.09044.

9 Dimitri P Bertsekas. *Nonlinear Programming*. Athena Scientific, 3 edition, 1999.

10 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

11 Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

12 Vitalik Buterin. Eip 150: Gas cost changes for io-heavy operations, 2016. URL: https://eips.ethereum.org/EIPS/eip-150.

13 Vitalik Buterin. Geth nodes under attack again, 2016. URL: https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/?st=itxh568s&sh=ee3628ea.

14 Vitalik Buterin. Transaction spam attack: Next steps, 2016. URL: https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/.

15 Vitalik Buterin. Easy parallelizability issue #648 ethereum/EIPs, 2017. URL: https://github.com/ethereum/EIPs/issues/648.

16 Vitalik Buterin. First and second-price auctions and improved transaction-fee markets, 2018. URL: https://ethresear.ch/t/first-and-second-price-auctions-and-improved-transaction-fee-markets/2410.

17 Vitalik Buterin. An incomplete guide to rollups, 2021. URL: https://vitalik.ca/general/2021/01/05/rollup.html.

18 Vitalik Buterin. Multidimensional eip 1559. Ethresearch, 2022. URL: https://ethresear.ch/t/multidimensional-eip-1559/11651.

19 Vitalik Buterin. Proto-danksharding FAQ, 2022. URL: https://notes.ethereum.org/@vbuterin/proto_danksharding_faq.

20 Vitalik Buterin. State of research: Increasing censorship resistance of transactions under proposer/builder separation (pbs), 2022. URL: https://notes.ethereum.org/@vbuterin/pbs_censorship_resistance.

21 Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. Eip-1559: Fee market change for eth 1.0 chain, 2019. URL: https://eips.ethereum.org/EIPS/eip-1559.

22 Vitalik Buterin and Martin Swende. Eip-2929: Gas cost increases for state access opcodes, 2020. URL: https://eips.ethereum.org/EIPS/eip-2929.

23 Vitalik Buterin and Martin Swende. EIP-2930: Optional access lists, 2020. URL: https://eips.ethereum.org/EIPS/eip-2930.

24 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). In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 2021. doi:10.1145/3477132.3483564.

25 Hao Chung and Elaine Shi. Foundations of transaction fee mechanism design. *arXiv preprint arXiv:2111.03151*, 2021.

26 Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

27 Theo Diamandis, Alex Evans, Tarun Chitra, and Guillermo Angeris. Dynamic pricing for non-fungible resources: Designing multidimensional blockchain fee markets. *arXiv preprint arXiv:2208.07919*, 2022.

28 Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM review*, 59(2):295–320, 2017.

29 Ethereum. Gas and fees, 2022. URL: https://ethereum.org/en/developers/docs/gas/.

30 Ethereum. Run a node, 2022. URL: https://ethereum.org/en/run-a-node/.

31 Matheus Ferreira, Daniel Moroz, David Parkes, and Mitchell Stern. Dynamic posted-price mechanisms for the blockchain transaction-fee market. In *Proceedings of the 3rd ACM conference on Advances in Financial Technologies*, pages 86–99, 2021.

32 John Forrest, Stefan Vigerske, Ted Ralphs, Lou Hafer, John Forrest, jpfasano, Haroldo Gambini Santos, Matthew Saltzman, Jan-Willem, Bjarni Kristjansson, h-i gassmann, Alan King, pobonomo, Samuel Brito, and to st. coin-or/clp: Release releases/1.17.7, January 2022. doi:10.5281/zenodo.5839302.

33 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, 2022. doi:10.48550/ARXIV.2203.06871.

34 Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

35 Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

36 Frank Kelly. Charging and rate control for elastic traffic. *European transactions on Telecommunications*, 8(1):33–37, 1997.

37 Kshitij Kulkarni, Theo Diamandis, and Tarun Chitra. Towards a theory of maximal extractable value i: Constant function market makers, 2022. arXiv:2207.11835.

38 Fuel Labs. GitHub - FuelLabs/fuel-specs: Specifications for the fuel protocol and the FuelVM, a blazingly fast blockchain VM, 2022. URL: https://github.com/FuelLabs/fuel-specs.

39 Stefanos Leonardos, Barnabé Monnot, Daniël Reijsbergen, Efstratios Skoulakis, and Georgios Piliouras. Dynamical analysis of the eip-1559 ethereum fee market. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 114–126, 2021.

40 Stefanos Leonardos, Daniël Reijsbergen, Daniël Reijsbergen, Barnabé Monnot, and Georgios Piliouras. Optimality despite chaos in fee markets. *arXiv preprint arXiv:2212.07175*, 2022.

41 Steven Low. A duality model of tcp and queue management algorithms. *IEEE/ACM Transactions On Networking*, 11(4):525–536, 2003.

42 Steven Low and David Lapsley. Optimization flow control. i. basic algorithm and convergence. *IEEE/ACM Transactions on networking*, 7(6):861–874, 1999.

43 Kamilla Nazirkhanova, Joachim Neu, and David Tse. Information dispersal with provable retrievability for rollups, 2021. doi:10.48550/ARXIV.2111.12323.

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

45 Polygon Team. Introducing avail by polygon, a robust general-purpose scalable data availability layer, 2021. URL: https://blog.polygon.technology/introducing-avail-by-polygon-a-robust-general-purpose-scalable-data-availability-layer/.

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

47 Daniël Reijsbergen, Shyam Sridhar, Barnabé Monnot, Stefanos Leonardos, Stratis Skoulakis, and Georgios Piliouras. Transaction fees on a honeymoon: Ethereum's eip-1559 one month later. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 196–204. IEEE, 2021.

48 Tim Roughgarden. Transaction fee mechanism design. *SIGecom Exch.*, 19(1):52–55, 2021. doi:10.1145/3476436.3476445.

49 Vikram Saraph and Maurice Herlihy. An empirical study of speculative concurrency in ethereum smart contracts, 2019. doi:10.48550/ARXIV.1901.01376.

50 Naum Zuselevich Shor. *Minimization methods for non-differentiable functions*, volume 3. Springer Series in Computational Mathematics, 1985.

51 Ertem Nusret Tas, Dionysis Zindros, Lei Yang, and David Tse. Light clients for lazy blockchains, 2022. _eprint: 2203.15968.

52 Jeffrey Wilcke. The ethereum network is currently undergoing a DoS attack, 2016. URL: `https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/`.

53 Anatoly Yakovenko. Sealevel, parallel processing thousands of smart contracts, 2020. URL: `https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192`.

54 Anatoly Yakovenko. Consider increasing fees for writable accounts issue #21883 solana-labs/solana, 2021. URL: `https://github.com/solana-labs/solana/issues/21883`.