Transaction throughput scalability strategies for Plutus smart contracts

Morgan Thomas / Platonic Systems

December 13, 2023

Abstract

Cardano's extended unspent transaction output (EUTXO) model is how the Cardano blockchain models state and resource ownership. The EUTXO model allows decentralized application (dapp) developers to model their business domain as UTXOs and their business logic as Plutus smart contracts. With on-chain dapp state stored as UTXOs, it can be manipulated by transactions according to rules determined by smart contracts. However, this situation gives rise to resource contention problems because each UTXO can only be consumed once. This paper reviews some known strategies and makes general recommendations for solving these resource contention problems for any particular use case.

1 Introduction

This paper may interest both engineers and non-engineers who wish to understand better the problem of designing dapps to be scalable when using Plutus smart contracts. The scalability of a software solution means, generally, how much time and money it takes to increase its capacity. Capacity can mean any number of things. Capacity can be measured in the number of users a computer system can support, the number of tasks it can perform in a given time frame, or the amount of data it can process and store.

For our purposes, it is helpful to think of scalability in terms of throughput. The throughput of a dapp, we can say, is the number of actions that users can perform in a given time period. For example, if we expect that in

a 60 second period, the dapp can validate and persist the results of up to 30 actions by up to 30 different users, then we would say that the dapp has an expected throughput of 30 transactions per minute, or in other words, 0.5 transactions per second.

If we can measure the maximum sustainable throughput of a dapp in terms of transactions per time period, then we can determine whether or not it will support the expected amount of usage. Visa, for example, reportedly processed an average of 1,700 transactions per second in 2019 (and its peak usage could therefore be assumed to be more than that). [1] For some dapps, such as a decentralized exchange (a DEX), it is conceivable that someday peak transaction volumes on the dapp will be higher than 1,700 transactions per second.

In theory, software solutions should be designed to support all of the scalability necessary to fully capitalize on the business opportunity within reason. In other words, a software solution should be designed to be capable, in principle, of meeting all of the world's demand for that solution.

We should bear scalability in mind when designing and implementing software solutions. There are many well-known techniques for doing so. One technique for analyzing scalability is asymptotic analysis. Asymptotic analysis determines how much of a finite resource an algorithm will consume. This is done by providing a function of inputs or properties (such as input sizes), which expresses an upper bound on resource consumption.

Asymptotic analysis of resource consumption is often expressed using big O notation.[2] For example, when we say that a resource requirement of a system is $O(n)$ in n the number of users, it means that we will have enough of the resource if we add an amount of the resource sufficient for one user each time we add one user. Or, if the resource requirement is $O(1)$ in the number of users, that means if we have enough of the resource to support a certain number of users, then we will continue to have enough of the resource no matter how many users we add.

An example of a scalability issue would be if a resource requirement was, let's say, no better than $O(2^n)$ in n the number of users. If this occurred, then each time we added one user, it may double the amount of the resource needed. A system that needs double the resource requirement every time a user is added can't scale to very many users. Asymptotic analysis helps to find suitable solutions to software problems by catching such scalability issues in the design phase, rather than catching them in the testing phase, where they are potentially far more expensive to fix.

Thus, to demystify the process of writing scalable Plutus dapps, I'll be analyzing scalability in terms of transaction throughput using asymptotic analysis.

2 The EUTXO model and Plutus smart contracts

The EUTXO model provides a means for modeling dapp state and ownership of resources which can be produced and consumed by dapps. In the EUTXO model, dapp developers use UTXOs to model their business domain on the blockchain, and they use smart contracts to model their business logic.

What is the EUTXO model? What is a UTXO, and what other concepts are involved in the EUTXO model? UTXO stands for unspent transaction output. A simple example of a UTXO is some money in a person's Cardano wallet. In the EUTXO model, UTXOs can contain data as well as cryptocurrency. The addition of the ability to store arbitrary data to UTXOs opens the door to modeling business domains on-chain and thus to creating dapps. The (E)UTXO model used by Bitcoin and Cardano can be contrasted with the account-based model used by Ethereum. [3, 4]

What is a UTXO, exactly, in Cardano, and what is a Plutus smart contract? Some precise answers to these questions will be helpful for the problem at hand.

2.1 General abstract terminology

In order to answer these questions concisely and precisely, I will be using a few simple concepts from category theory. Category theory is a general language of thought used by mathematicians, engineers, and scientists. Category theory allows me to speak at a level of abstraction which conveys this information in a way that is precise but not tied to the less relevant details of how Plutus is actually implemented as Haskell software.

For this explanation of the EUTXO model, I will be using the following concepts from category theory: category, type or object, function type or exponential object, isomorphism, (Cartesian) product, and coproduct or sum. For a rigorous yet gentle introduction to category theory, I would recommend Awodey [5], but for our purposes, a non-rigorous yet brief introduction to these concepts will suffice.

A category consists of a set of objects and a set of arrows or morphisms between those objects. An arrow has a domain object (which the arrow is "from"), and a codomain object (which the arrow is "to"). Additionally, arrows have a composition relation: if a, b, c are objects in a category C and $f : a \to b$ and $g : b \to c$ are arrows, then there is an arrow $g \circ f : a \to c$ called the composition of g with f. This composition relation satisfies the associativity property $f \circ (g \circ h) = (f \circ g) \circ h$. Additionally, for each object a in C, there is a morphism $\mathrm{id}_a : a \to a$ such that for all morphisms $f : a \to b$, $f \circ \mathrm{id}_a = f$, and for all morphisms $g : b \to a$, $\mathrm{id}_a \circ g = g$. If these conditions are satisfied, then we can consider the tuple $(C, \hom(C), \circ, \mathrm{id})$ to be a category, where C is the set of objects, $\hom(C)$ is the set of morphisms, $\circ$ is the composition relation, and $\mathrm{id} : C \to \hom(C)$ is the set of identity morphisms indexed by the objects of the category.

Depending on the category, the objects of the category are sometimes called types. That is not an accurate description of what objects are in all categories, but for the categories we are concerned with, it is. Thus when you hear "object" in this context, you can think "type." Or, if you prefer to think of the mathematical concept of "set," that is also accurate; the objects of the categories we are concerned with are for all intents and purposes sets.

If you like, you can think of the category of Haskell types as the category we are working with, where the morphisms are partial computable Haskell functions. If this helps you, then you can think of it that way, but we are not employing specific properties of that category, beyond basic properties satisfied by many other categories, like the existence of products, coproducts, and exponential objects (to be explained momentarily).

An isomorphism in a category is a morphism $f : a \to b$ such that there is a morphism $g : b \to a$ such that $f \circ g = \mathrm{id}_b$ and $g \circ f = \mathrm{id}_a$. You can think of an isomorphism as an invertible transformation: one which can be undone, giving you back the same thing you started with, belonging to the same type you started with, such that every thing in the type you started with can be inverted in such a fashion. Adding three to an integer is reversible; you can simply subtract three. Adding three to an integer is an isomorphism. Multiplying an integer by three is not an isomorphism, because not every integer is a multiple of three; thus multiplying an integer by three does not have an inverse which is defined on all integers. Dividing an integer by three is also not an isomorphism, because while every integer is the result of dividing an integer by three, not every integer can be divided by three with the result being an integer.

We write $a \cong b$ to denote that a is isomorphic to b, or in other words, there exists an isomorphism $f : a \to b$.

A (Cartesian) product of two objects a and b in a category is an object $a \times b$ which is like the type of ordered pairs where the first element is of type a and the second element is of type b. I will not give a precise definition of products, but see [5] if you want one. You can also have products of more than two sets, like $a \times b \times c \times d$, which is a type where the values are 4-tuples each consisting of an a, a b, a c, and a d.

A coproduct (or sum) of two objects a and b in a category is an object $a \oplus b$ which is like the disjoint union of a and b. A value of type $a \oplus b$ is either a value of type a or a value of type b (but not both). A value of type $a \oplus a$ is either a "left a" or a "right a." If x is a value of type a, then x is not a value of type $a \oplus a$, but "left x" and "right x" are values of type $a \oplus a$. Thus $a \oplus a$ is not isomorphic to a, but $a \oplus a$ is in fact isomorphic to $a \times \mathbf{2}$, where $\mathbf{2}$ is a type with exactly two values: $\mathbf{2} = \{0, 1\}$. I will not give a precise definition of coproducts but [5] has one.

In Haskell, we can think of products and coproducts as equivalent to regular algebraic data types (as opposed to, say, generalized algebraic data types). A regular algebraic data type consists of zero or more constructors, each of which takes zero or more arguments, each of which has a type. A regular algebraic data type is isomorphic to a product of coproducts, where each constructor can be replaced with a product and the whole algebraic data type with a sum of the products resulting from its constructors.

A function type or exponential object is essentially a type of morphisms. We can write the function type $a \to b$ in other notation as the exponential object b^a, which is the same thing. In the category of sets, b^a is the function space (i.e., the set of functions) from a to b. In the category of Haskell types, $a \to b$ is a function type, the type of functions from $a \to b$. In each category, we can call it an exponential object and write it $a \to b$ or b^a. I will not give a precise definition of exponential objects but [5] has one.

That is all of the category theory terminology we will need for this discussion.

2.2 Defining UTXOs (up to isomorphism)

Now let us return to our main questions for this section. What is a UTXO, and what is a smart contract? These questions have precise and specific answers in the language of category theory. Here is my main claim regarding

what a UTXO is:

$$\text{UTXO} \cong \text{Address} \times \text{Value} \times \text{Maybe Datum} \tag{1}$$

In other words, the type of UTXOs is isomorphic to a product of a Cardano Address, a Value, and a Datum.

The Address is the address of the owner. Each UTXO contains zero or more kinds of cryptocurrency in it, in some amounts; that is the Value. Each UTXO may contain arbitrary data in it, such as state data for a dapp; that is the Datum.

In order to explain this claim about what a UTXO is, I will provide some context. I will provide this context in the form of further category theory claims and context for those claims. This process will give us a concrete and precise and verifiably accurate concept of what a UTXO is in Cardano.

The following claims are sourced from the source code of Plutus: [6]

$$\text{Value} \cong \text{Map CurrencySymbol (Map TokenName Integer)} \tag{2}$$

This is true by definition of Value in Plutus.V1.Ledger.Value. What are CurrencySymbol and TokenName?

$$\text{CurrencySymbol} \cong \text{BuiltinByteString} \tag{3}$$

$$\text{TokenName} \cong \text{BuiltinByteString} \tag{4}$$

These are also true by definitions in Plutus.V1.Ledger.Value. And what is BuiltinByteString?

$$\text{BuiltinByteString} \cong \text{ByteString} \tag{5}$$

A BuiltinByteString is isomorphic to a sequence of zero or more bytes. That tells us what CurrencySymbol and TokenName are isomorphic to, but not what they mean. Together they represent an asset class:

$$\text{AssetClass} \cong \text{CurrencySymbol} \times \text{TokenName} \tag{6}$$

This is true by definition of AssetClass in Plutus.V1.Ledger.Value.

Overall, a Value is isomorphic to a map from zero or more asset classes to their corresponding integer quantities. Thus a Value represents some

cryptocurrency, in one or more asset classes. An asset class is a fungible type of cryptocurrency, such as for example, ADA (Cardano).

What is an Address?

$$\text{Address} \cong \text{Credential} \times \text{Maybe StakingCredential} \tag{7}$$

$$\text{Maybe StakingCredential} \cong \mathbf{1} \oplus \text{StakingCredential}, \tag{8}$$

where $\mathbf{1} = \{0\}$ is a type with only value of that type. In this context the value 0 represents the absence of a staking credential. These are true by the definition of Address in Plutus.V1.Ledger.Address.

$$\text{Credential} \cong \text{PubKeyCredential} \oplus \text{ScriptCredential} \tag{9}$$

$$\text{PubKeyCredential} \cong \text{PubKeyHash} \tag{10}$$

$$\text{ScriptCredential} \cong \text{ValidatorHash} \tag{11}$$

These are true by the definition of Credential in Plutus.V1.Ledger.Credential.

$$\text{PubKeyHash} \cong \text{BuiltinByteString} \tag{12}$$

This is true by the definition of PubKeyHash in Plutus.V1.Ledger.Crypto.

$$\text{ValidatorHash} \cong \text{BuiltinByteString} \tag{13}$$

This is true by the definition of ValidatorHash in Plutus.V1.Ledger.Scripts.

What is a StakingCredential?

$$\text{StakingCredential} \cong \text{StakingHash} \oplus \text{StakingPtr} \tag{14}$$

$$\text{StakingHash} \cong \text{Credential} \tag{15}$$

$$\text{StakingPtr} \cong \text{Integer} \times \text{Integer} \times \text{Integer} \tag{16}$$

These are true by definition of StakingCredential in Plutus.V1.Ledger.Credential.

What is a Datum? In dapp development, ideally, it is a type defined by the dapp developer to model the possible states of this UTXO. From an

implementation standpoint, it needs to be represented on-chain as Plutus Core Data. Thus we can say, from this standpoint:

$$\text{Datum} \cong \text{Data} \tag{17}$$

And what is Data?

$$\text{Data} \cong \text{Constr} \oplus \text{Map} \oplus \text{List} \oplus \text{Integer} \oplus \text{ByteString} \tag{18}$$

$$\text{Constr} \cong \text{Integer} \times \text{Data} \tag{19}$$

$$\text{Map} \cong [\text{Data} \times \text{Data}] \tag{20}$$

Here $[a]$ denotes the type of a sequence of zero or more elements of type a.

$$\text{List} \cong [\text{Data}] \tag{21}$$

These are true by definition of Data in PlutusCore.Data.

2.3 UTXOs in the Cardano EUTXO literature

The previous section provides a concrete description of what I am claiming a UTXO is, up to isomorphism:

$$\text{UTXO} \cong \text{Address} \times \text{Value} \times \text{Maybe Datum} \tag{22}$$

What further context supports this claim? Firstly, review of the literature published by IOHK, and secondly, analysis of the isomorphisms which I will state in Subsection 2.4. Let's look at the literature.

> To maintain the machine state, we extend UTXO outputs from being a pair of a validator ρ and a cryptocurrency value value to being a triple $(\rho, \text{value}, \delta)$ of validator, value, and a datum δ, where δ contains arbitrary contract-specific data.

Source: Chakravarty et al, "The Extended UTXO Model," IOHK, 2021, Section 2, page 3. [8]

> How does the EUTXO model extend UTXO?
>
> By adding custom data to outputs (in addition to value), and by allowing for more "locks" and "keys" deciding under which

> condition an output can be unlocked for consumption by a transaction. In other words, instead of just having public keys (hashes) for locks and corresponding signatures serving as "keys", EUTXO enables arbitrary logic in the form of scripts. This arbitrary logic inspects the transaction and the data to decide whether the transaction is allowed to use an input or not.

Source: Sanchez, "Cardano's Extended UTXO accounting model – built to support multi-assets and smart contracts (part 2)." IOHK, 2021. [3, 4]

2.4 Defining smart contracts (up to isomorphism)

Defining smart contracts provides further context in which we can justify the principal claim of this section about a UTXO is, that is, UTXO $\cong$ Address $\times$ Value $\times$ Maybe Datum. The following isomorphisms define (up to isomorphism) what a smart contract is.

I am considering a smart contract to be equivocable with the validator of the smart contract. A smart contract, considered as a deliverable software solution, consists of both on-chain code which is compiled to Plutus Core and run on the blockchain, and on the other hand, off-chain code which runs as additional components of the dapp. The same code can also run both on-chain and off-chain.

The on-chain portion of a smart contract consists of a validator. This validator is what actually defines the rules of the contract. The contract will allow whatever transactions satisfy the validator. A deliverable smart contract also includes off-chain code which constructs transactions which will satisfy the validator and posts them to the blockchain. However, users of the smart contract are not required to use the dapp developers' off-chain code in order to post transactions to the blockchain.

Any way of creating a valid transaction with a smart contract may be used to cause a transaction with a smart contract to occur on-chain. That is why it is true that the validator alone defines the actual rules of the contract. That is why I am equivocating in this section between the smart contract and the validator. If you do not agree with my equivocation, then you may ignore that part of what I am saying and consider it just as a definition of what a validator is (up to isomorphism), and this still supports the main claim about what a UTXO is when we get to Subsection 2.5.

$$\text{SmartContract} \cong \text{Validator} \tag{23}$$

That is the conclusion of the argument above.

$$\text{Validator} \cong \text{Datum} \to \text{Redeemer} \to \text{ScriptContext} \to \text{Bool} \tag{24}$$

Source: definition of ValidatorType in Ledger.Scripts.Typed.Validators. [6]

Redeemer is like Datum in the sense that from the dapp developer's perspective, we would like it to be a type which the dapp developer defines, which reflects something like the type of what different interactions users may have with the dapp. On the other hand, from an implementation standpoint, like Datum, Redeem ultimately needs to be put into an on-chain form where it is untyped Data.

$$\text{Redeemer} \cong \text{BuiltinData} \cong \text{Data} \cong \text{Datum} \tag{25}$$

Sources: definition of Redeemer in Plutus.V1.Ledger.Scripts, and definition of BuiltinData in PlutusTx.Builtins.Internal, and Isomorphism 17.

We should not, taking Isomorphism 25 out of context, conclude that from a dapp developer's perspective, it is true that Redeemer $\cong$ Datum. From a dapp developer's perspective, these are (hopefully) not their underlying on-chain untyped representations, but semantically meaningful types.

The following are true by the definitions in Plutus.V1.Ledger.Contexts, Plutus.V1.Ledger.Scripts, Plutus.V1.Ledger.Tx, and Plutus.V1.Ledger.TxId. [6]

$$\text{ScriptContext} \cong \text{TxInfo} \times \text{ScriptPurpose} \tag{26}$$

$$\begin{aligned} \text{TxInfo} \cong \quad & [\text{TxInInfo}] \times [\text{TxOut}] \times \text{Fee} \times \text{Mint} \\ & \times [\text{DCert}] \times \text{Withdrawals} \times \text{ValidRange} \\ & \times \text{Signatories} \times [\text{DatumHash} \times \text{Datum}] \times \text{TxId} \end{aligned} \tag{27}$$

$$\text{Fee} \cong \text{Value} \tag{28}$$

$$\text{Mint} \cong \text{Value} \tag{29}$$

$$\text{Withdrawals} \cong [\text{StakingCredential} \times \text{Integer}] \tag{30}$$

$$\text{ValidRange} \cong \text{POSIXTimeRange} \tag{31}$$

$$\text{Signatories} \cong [\text{PubKeyHash}] \tag{32}$$

$$\text{TxInInfo} \cong \text{TxOutRef} \times \text{TxOut} \tag{33}$$

$$\text{TxOutRef} \cong \text{TxId} \times \text{Integer} \tag{34}$$

$$\text{TxId} \cong \text{BuiltinByteString} \tag{35}$$

$$\text{TxOut} \cong \text{Address} \times \text{Value} \times \text{Maybe DatumHash} \tag{36}$$

This Isomorphism 36 is a good one to be aware of for its relevance to the UTXO definition problem.

$$\text{DatumHash} \cong \text{BuiltinByteString} \tag{37}$$

A DCert is a digest of certificates. The following come from Plutus.V1.Ledger.DCert.

$$\begin{aligned}\text{DCert} \cong\ & \text{DCertDelegRegKey} \oplus \text{DCertDelegDeRegKey} \oplus \text{DCertDelegDelegate} \\ & \text{DCertPoolRegister} \oplus \text{DCertPoolRetire} \oplus \text{DCertGenesis} \oplus \text{DCertMir}\end{aligned} \tag{38}$$

$$\text{DCertDelegRegKey} \cong \text{StakingCredential} \tag{39}$$

$$\text{DCertDelegDeRegKey} \cong \text{StakingCredential} \tag{40}$$

$$\text{DCertDelegDelegate} \cong \text{StakingCredential} \times \text{PubKeyHash} \tag{41}$$

$$\text{DCertPoolRegister} \cong \text{PoolId} \times \text{PoolVFR} \tag{42}$$

$$\text{PoolId} \cong \text{PubKeyHash} \tag{43}$$

$$\text{PoolVFR} \cong \text{PubKeyHash} \tag{44}$$

$$\text{DCertPoolRetire} \cong \text{PubKeyHash} \times \text{Integer} \tag{45}$$

$$\text{DCertGenesis} \cong \mathbf{1} \tag{46}$$

$$\text{DCertMir} \cong \mathbf{1} \tag{47}$$

The following are true by the definition of ScriptPurpose in Plutus.V1.Ledger.Contexts.

$$\text{ScriptPurpose} \cong \text{Minting} \oplus \text{Spending} \oplus \text{Rewarding} \oplus \text{Certifying} \tag{48}$$

$$\text{Minting} \cong \text{CurrencySymbol} \tag{49}$$

$$\text{Spending} \cong \text{TxOutRef} \tag{50}$$

$$\text{Rewarding} \cong \text{StakingCredential} \tag{51}$$

$$\text{Certifying} \cong \text{DCert} \tag{52}$$

That completes the precise description of what a validator (or a smart contract) is, up to isomorphism.

2.5 Further contextualization of UTXO $\cong$ Address $\times$ Value $\times$ Maybe Datum

What is the relevance of these isomorphisms describing validators to the question of what a UTXO is and the justification of my answer? Consider the following:

$$\begin{aligned}[\text{Address} \times \text{Value} \\ \times \text{Maybe Datum}] \quad &\cong \quad [\text{DatumHash} \times \text{Datum}] \times \\ &\quad\ [\text{Address} \times \text{Value} \times \text{Maybe DatumHash}]\end{aligned} \tag{53}$$

In the right hand side, [DatumHash $\times$ Datum] is a relation where a datum is only associated with its hash, and any DatumHash occurring in the relation [Address $\times$ Value $\times$ Maybe DatumHash] must occur in the datum hash to datum relation. Since the DatumHash is a pure function of the Datum, these two sides of the isomorphism are just two ways of expressing the same data.

According to Isomorphism 27, TxInfo is a product which includes [TxOut] and [DatumHash $\times$ Datum]. TxOut $\cong$ Address $\times$ Value $\times$ Maybe DatumHash according to Isomorphism 36. Therefore TxInfo contains all the information needed to produce a [UTXO] by following Isomorphism 53.

3 Modeling maximum throughput for Plutus smart contracts

We are interested in modeling the maximum throughput for a Plutus smart contract. We are measuring throughput in the number of interactions with the contract which can be performed in a time interval. We are considering an interaction to have been performed when its results have been included in the blockchain.

For the sake of an example, suppose that the throughput of a contract is limited in the following way: only one interaction can be performed per block. Then:

$$\text{max throughput of contract} \leq \text{throughput of block creation} \tag{54}$$

Say we assume for the sake of estimation, consistent with [9], and probably fairly realistic for the near future, that

$$\text{throughput of block creation} = 3 \text{ blocks per minute} \tag{55}$$

Then for this example,

$$\text{max throughput of contract} \leq 3 \text{ interactions per minute} \tag{56}$$

This example is a realistic one. It models any scenario where there is one UTXO which every interaction with the contract must consume. This is true of a contract which has a single instance and a single state datum UTXO which every interaction with that instance must consume.

For some applications, a maximum throughput of 3 interactions per minute per contract may be fine. It is probably fine, for example, if the contract is expected to have a maximum of three users and about ten contract interactions over the course of its lifetime. This may be a realistic throughput expectation for some contracts.

On the other hand, some applications require much higher throughput than 3 interactions per minute. Since each UTXO can only be consumed by one transaction on a block, resource contention becomes a consideration in contract design specifically with respect to UTXOs. Dapp users expect to be able to perform actions on the dapp in a timely fashion. Not taking UTXO contention into consideration in contract design may violate this expectation.

In order to study the maximum throughput of contract designs with higher maximum throughput, we should consider what happens if more than one UTXO is available to be consumed when a user interaction needs to consume a UTXO.

For a slightly more complex example, let us consider a scenario where each interaction with the dapp must consume exactly one UTXO out of a pool of n UTXOs. Each time an interaction is performed, it consumes one UTXO out of the pool. All of the UTXOs in the pool are equivalent for all intents and purposes, but importantly, they are distinct from each other. Thus, when a user wishes to perform an interaction, their off-chain code randomly selects a UTXO from the pool. The interaction consumes that UTXO and outputs another UTXO to replace it, which may be used by the next person wanting to interact with the dapp. It is still possible for conflicts to occur, where users select the same UTXO to use via the random decentralized selection process. However, this is less likely to occur than in the case where there is only one UTXO available for all interactions. The result of a conflict may be a user's transaction failing go through and more latency is added to the interaction.

We are interested in quantifying the maximum throughput in interactions per time period for such a system. It is clear that

$$\text{max transaction throughput} \leq n \cdot \text{throughput of block creation} \tag{57}$$

This is because in each block, at most n transactions can consume a UTXO, under the given scenario.

For example, with $n = 3,000$, i.e. 3,000 UTXOs in the pool, no more than 9,000 interactions per minute can be performed. To exceed the average

throughput of Visa, which is 1,700 transactions per second, $n = 35,000$ could be sufficient.[1]

Now let us suppose that for some reason we want the same throughput with a lower number of UTXOs. Let us consider a slightly more complex scenario designed to address this consideration. Let us suppose that one transaction on the blockchain, which consumes one UTXO out of the pool of n UTXOs, can process k distinct interactions by up to k distinct users. Then

$$\text{max interaction throughput} \leq n \cdot k \cdot \text{throughput of block creation} \tag{58}$$

Then to exceed the throughput of Visa, $n = 350$ and $k = 100$ could, for example, suffice.

These are some examples of varying complexity, ultimately all relatively simple, of how we could put upper bounds on the interaction throughput of a dapp based on modeling its UTXO resources and requirements. This simple modeling approach can inform the design of dapps given assumptions about their interaction throughput requirements.

In order to determine the actual maximum throughput of a dapp, we need to use empirical measurements. In order to estimate the maximum throughput of a dapp design, we can use analysis of known limiting factors. The limiting factors on an app's interaction throughput are those factors which cause constraints of the form interaction throughput $\leq l$. We can estimate an app design's max throughput by estimating the constraints caused by the known limiting factors, in which case

$$\begin{aligned} &\text{estimated max interaction throughput} = \\ &\min\{l \mid (\text{interaction throughput} \leq l) \text{ is a constraint} \\ &\ \text{caused by a known limiting factor.}\} \end{aligned} \tag{59}$$

A complete analysis of the scalability of a design should estimate whether or not all known limiting factors on all relevant scalability measurements (not just max interaction throughput) are going to result in an acceptable potential for scalability of the design. When known limiting factors are expected

[1] Since the maximum throughput of Visa can be assumed to be greater than the average throughput, but I do not know it, I cannot compare this figure to the maximum throughput of Visa.

to potentially impact scalability, we should try to analyze the options for addressing these scalability challenges and determine a good path forward.

However, the scope of this paper is limited to consideration of max interaction throughput for a Plutus-based dapp as a limiting factor on scalability.

4 Design considerations for throughput scalability strategies for Plutus smart contracts

What is a throughput scalability strategy for Plutus smart contracts? It is a design pattern that can be used to optimize smart contract designs for scalability with respect to interaction throughput. We are going to look at design patterns which point to an infinite creative potential in designing throughput scalability strategies for Plutus smart contracts. There is an infinite number of ways of combining, specializing, and applying the design patterns we are going to review. Each of these can be tested empirically to measure its throughput, but this is not something we can study empirically over the whole space of possible applications, because that space is infinite. Also, I am not providing a precise mathematical definition of what a throughput scalability strategy or a design pattern is. As such, I cannot speak precisely about all throughput scalability strategies. I am merely pointing to some possible approaches which project designers can take and offering some cursory analysis of the pros and cons of these approaches.

When considering applying any throughput scalability strategy, we ought to look at how it impacts not just the factor we are intending to impact which is interaction throughput. We ought to look also at how the throughput scalability strategy impacts all other factors of interest, including but not limited to, correctness, cost, complexity, security, maintainability, sustainability, and user experience. At first, each problem should be considered as a special case and a case study, and as we learn more about how different scalability strategies play out in different applications, then we can begin to rely on inductive generalizations to estimate the consequences of different design decisions.

How do we choose which designs to test given little to no available empirical data on the results of different decisions? There is some cost to creating a design to a level of completeness which is testable. As such, we can only test a finite number of designs. Perhaps, given resource constraints, at first we can only test one design, and it needs to work with minimal modifica-

tions in order to meet anticipated project timelines. Under such constraints, we must choose a design based on a priori reasoning, and the design we choose may likely be chosen partly because it is amenable to estimation of its consequences based on a priori reasoning.

Here are the basic considerations I think are most relevant to study (a priori) for estimating the results of applying throughput scalability strategies for Plutus dapps: correctness, complexity, security, user experience, and of course, scalability. We have already seen simple methods for providing estimates of the scalability of Plutus dapps assuming UTXO contention as a limiting factor. Now let us look at the question, how should we look at the other considerations besides scalability when we look at applying a throughput scalability strategy for a Plutus dapp?

4.1 Correctness

A throughput scalability strategy needs to be correct. Very often we might have in mind an idea of what our contract's rules are if it is written as a Plutus contract with no scalability strategy. Then we might wish to apply a scalability strategy as a semantics preserving transformation on the nonscaled contract. This may not always be something we can find a way to do. It may be necessary to change the semantics of the contract in order to apply a scalability strategy to it. Applying a scalability strategy may change the semantics of a contract in subtle and unexpected ways (bugs).

A scalability strategy, viewed as a semantics preserving transformation or some approximation thereof, takes a contract with sequential semantics only and turns it into a contract which preserves the same semantics (or approximately does so) for sequential use cases, while also opening up use cases where parallelism causes higher throughput potential. The generalization of the sequential semantics to the parallel use case has the potential to violate the intentions of the designer. Invariants of the contract, properties which are supposed to be satisfied by each successive state of the blockchain, which is true in the serial contract, may not always hold true in the parallel use cases. If those invariants are expected and required, then it is a bug.

Generally speaking, bugs resulting from parallelizing a sequential design are concurrency issues, such as deadlocks and race conditions. The available strategies for parallelizing a sequential UTXO data model and contract reflect the broader marketplace of ideas for writing software with parallel and concurrent behavior, including but not limited to these. We can use threads

with isolated states. We can use buffers. We can use message passing concurrency. We can use locks. The use of these mechanisms can lead to the classical types of bugs that can come from use of these mechanisms.

Other more novel classes of bugs may arise from the complex nature of time on the Cardano blockchain, which is different from how most engineers are used to thinking about time in computers, and which has its own unique challenges. Time in Cardano advances linearly, but it cannot be measured in terms of POSIX or UTC time; a point in time on the Cardano blockchain is not a time in the usual sense, but a slot. A slot is an integer which increases linearly with each block. A slot does not denote a specific point in time in the usual sense; rather, it denotes an index in a sequence of blocks. But, this is what we think of as time in on-chain code. On-chain code thinks of the current slot as the current time. In reality, on-chain code is executed at multiple times and places, and thus, the question of when it is executing has no well-defined answer and the behavior of on-chain code cannot depend on the answer to that question. It is possible that the complicated nature of time on chain may interact with concurrency mechanisms in ways which have yet to be discovered which will lead to new classifications of bugs being published.

To the extent that applying a scalability strategy *does not* change the intended semantics of the contract for sequential use cases, we can say that the correctness issues that arise are limited to issues with the approach to generalizing to a parallel semantics.

To the extent that applying a scalability strategy *does* change the intended semantics of a contract, which may happen in some cases, this may impact the security, economics, and user experience, and it may ultimately not be correct, but that has to be analyzed on a case by case basis.

4.2 Complexity

Simpler is almost always better as long as it is correct. Complexity impacts on other design considerations of interest, including security, cost, correctness, maintainability, sustainability, and user experience, both directly and indirectly, generally always in the wrong direction. Complexity is necessary in order to surmount the inherent complexity of solving the problem, but minimizing complexity tends to help all aspects of the project, as long as it is not pursued to an excessive extent.

Scalability strategies for Plutus dapps tend to impact complexity in the

wrong (adverse) direction and this is not ideal. It also means that I give points to solutions which have less impact on the complexity of the contracts and perhaps more significantly the complexity of validating the correctness of the dapp.

4.3 Security

Security is something that we can look at from several different perspectives when it comes to UTXO data models and smart contracts. Just to enumerate a few, in no particular order:

1. One aspect of security is data privacy, where sensitive information owned by a person is not disclosed unexpectedly.

2. One aspect of security is data retention, where valuable information owned by a person is not lost or destroyed by accident.

3. One aspect of security is platform integrity, where the rules of the contract are performed in an intended manner without corruption or data loss.

4. One aspect of security is contract security, where the rules of the contract (both as intended and as actually implemented) are well defined and fair, and equitable and provide proper avenues of recourse to contract participants in courses of events that may cause harm or loss. This does not mean that the contract must guarantee that all participants are protected from loss, but it should give all contract participants proper recourse and contractually guaranteed rights even in courses of events which may cause harm or loss. The contract should have clearly defined rules which cannot be changed arbitrarily by the contract maintainer, as opposed to giving the contract maintainer the unlimited ability to change the contract in any way they choose.

Of these different aspects of security, only contract security (4) seems to me to be something impacted by the throughput scalability strategies we will example. Throughput scalability strategies create the potential for contract bugs and unintended economic consequences from parallel use cases. We can look at this from the perspective of imagining bad actors trying to exploit a system and what they would do, or we can look at this from the more neutral

perspective of trying to imagine economic actors attempting to pursue their incentives and what they would rationally do. Any economic vulnerability in the system (any economic dynamic which makes the system unsustainable) points to a fault in the system as opposed to a fault in the intentions of an actor who profited from the vulnerability. The intentions may be at fault as well in some cases, but that is no concern here. From this point of view, we can look at the contract maintainers also as rational economic actors and look to create a contract design which even the maintainers cannot render unsustainable by pursuing their individual economic incentives, without making assumptions about anybody's motives or intentions such as by assuming they can be trusted to do the right thing.

4.4 User experience

I think that normally I would want the impact of a throughput scalability strategy on user experience to be none. The only desired impact is that many users can interact with the dapp at the same time. Outside of that, I want there to be no impact on the user experience, at least, in the cases I have thought about so far.

4.5 Scalability

At a high level of abstraction, in order to achieve high transaction throughput, we need to avoid UTXO contention. If there is no UTXO contention, then every interaction can be processed with minimal latency. This means that we have to consider in tandem the problem of creating a UTXO data model of our business domain, together with the problem of creating and implementing a concurrent and parallel semantics of our contract business logic, and we have to consider both of these aspects of the solution with respect to the scalability properties they result in.

We can think of the state of our dapp, at a point in time on-chain, as a set of UTXOs. This is (the value of) the UTXO data model (at that point in time). We can ask questions about this data model which are normal to ask of a data model, such as these. Is the data model normalized (in any interesting sense)? What does it mean for a UTXO data model to be normalized?

I think that the following questions are good ones to ask about a data model to get a sense of how UTXO contention may impact scalability metrics.

What UTXOs exist in the data model? What UTXOs does a given interaction require in order to happen? How many UTXOs are consumed per interaction (minimum and maximum)? What is the expected frequency of another interaction at approximately the same time requiring the same UTXO, under various assumptions such as assumptions about interaction load?

If the state of one UTXO changes, does that invalidate state stored in any other UTXO? In other words, are there data dependencies between values stored in datums across UTXO boundaries? If such dependencies exist, then how is the invalidation of invalid data in UTXOs caused and enforced, and recovered from?

One interesting concept of normalization for UTXO data models may be this. Let's say that a UTXO data model is "UTXO-normalized" if and only if there are no data dependencies which cross UTXO boundaries, in the sense that if a UTXO is consumed and another UTXO is output to replace it, then that does not invalidate any data stored in any other UTXO not consumed by the same transaction.

UTXO-normalization could be helpful for scalability, for the reason that if, when a UTXO gets consumed, some other UTXO not consumed at the same time must be updated, this increases the number of UTXOs ultimately impacted by the interaction, with the potential for conflict with other interactions therefore increasing. If throughput is decreased by conflicts over UTXOs, and UTXO-normalization means that conflicts over UTXOs only occur when two transactions try to consume the same UTXO, then UTXO-normalization means that conflicts over UTXOs are both automatically prevented (causing some transactions to fail) and less likely to occur than they could be if UTXOs not consumed by a transaction were invalidated by it.

This property of UTXO-normalization seems like a desirable property for limiting the complexity of the system by avoiding data dependencies which cross UTXO boundaries. This would seem to ease the transition to scalability, on the face of it, by making the system easier to analyze. It also might increase scalability, as outlined the preceding paragraph. It also turns out to be a fairly stringent constraint in some ways.

It is true that a UTXO data model is UTXO-normalized if there is always exactly one UTXO in the data model and every interaction must consume that UTXO and produce its replacement. This is what mathematicians would call trivially true, because if there is only one UTXO and it is consumed by each transaction, then it cannot happen that a transaction would invalidate

data stored on a UTXO in the data model not consumed by the transaction, because there is no UTXO in the data model not consumed by the transaction, regardless of which transaction we may be talking about.

Although the UTXO-normalization property is trivially satisfied by any UTXO data model with exactly one UTXO consumed and produced by each interaction, it becomes a more stringent constraint when we are applying it to data models with more than one UTXO. It implies that when an interaction interacts with a piece of the dapp state, no other interaction interacts with the same exact piece of the dapp state in the same block. If the UTXO-normalization property is not satisfied, then it may be the case that two interactions interact with the same exact piece of the dapp state in the same block, and this may lead to an inconsistency that may (or may not) later be resolved.

5 Known throughput scalability strategies for Plutus smart contracts

Now it is time to look at some concrete examples of throughput scalability strategies for Plutus smart contracts.

For examples of applying these scalability strategies, we will use one running example: a DEX. A decentralized exchange (DEX) lets users trade on-chain assets using smart contracts. In a DEX, money changes hands peer to peer, asynchronously, mediated by a smart contract as opposed to a centralized broker or exchange. We will look at different strategies for implementing a DEX using Plutus smart contracts. These strategies have different scalability properties a priori, individually and in combination with each other. We will look at that. We will also look at other pros and cons of these strategies and combinations thereof.

5.1 Consume no UTXOs

IOHK [7] writes:

> Another way to run Plutus scripts on the ledger is by creating tokens with a custom minting policy. From a scalability perspective, minting scripts are great because they do not consume a script input. They aren't subject to UTXO congestion on script

> outputs, while allowing us to run a script in the transaction that produces the tokens. Seeing the token on the ledger is therefore evidence that the minting policy script has been executed successfully (as opposed to seeing a script output on the ledger, which can be produced without running any scripts at all). Whenever we need to run a Plutus script in our application we should ask ourselves if we can make this script a minting policy, and only use validators if we absolutely have to store some information or crypto currency value in a transaction output.

If a smart contract interaction consumes no UTXOs, then it is not subject to UTXO contention. UTXO contention is not a limiting factor for throughput scalability with respect to those interactions which consume no UTXOs. For this type of interaction, the UTXO consumption is O(1) in the interaction throughput (it is constant zero). So, if you can design your contract so that all interactions requiring massive throughput can be done without consuming UTXOs, then you can use this strategy alone (consuming no UTXOs) as your scalability strategy.

Can we apply this strategy to a DEX? Yes, perhaps for example in the following way. We can use an order book data model, where an open limit order is a UTXO, and creating an open order consumes no UTXO. Then we provide a matching engine, which examines the on-chain UTXOs and submits transactions that perform trades against matching open orders.

In this example, the matching transactions consume UTXOs, and there is potential for UTXO contention there. One simple way to avoid this is to make the matching API a single-user API called by a centralized algorithm. This would make the functioning of the DEX less centralized, but potentially free of UTXO contention.

5.2 Single threaded state machine

The single threaded state machine is a Plutus contract design pattern where each interaction with the contract consumes the current contract state UTXO and outputs the next contract state UTXO.

This pattern results in a contract which can support at most one interaction per block, which we are estimating to work out to about three interactions per minute. This is, presumably, enough for some contracts.

For dapps with scalability concerns, the single threaded state machine pattern alone will not suffice, but it may suffice in combination with other patterns we are discussing here.

5.3 Buffering

Buffering is a design pattern where a single Plutus transaction processes more than one interaction with a contract. These interactions, all the interactions performed by a single Plutus transaction in the buffering design pattern, can be performed by as many distinct users as there are interactions.

The idea of the buffer design pattern is that by a centralized buffering process, some of the requests users make are put into a buffer, up to the maximum capacity of a Plutus transaction, and are submitted to the blockchain by a transaction that has to be signed by the centralized controller. This ensures that there is no UTXO contention by having one UTXO and one UTXO consumer (the centralized controller). However, the reliance on centralized control seems unappealing if the goal is to create a decentralized contractual protocol.

This design pattern creates the constraint:

$$\text{max interaction throughput} \leq k \cdot \text{block creation throughput} \tag{60}$$

where k is the maximum number of interactions that can be stored in a single Plutus transaction for this contract.

Like the single threaded state machine pattern, the buffering pattern imposes an $O(1)$ constraint on max interaction throughput, which means there is some constant limiting factor on interaction throughput which we cannot exceed by any means using this strategy alone. The exact value of that constant limiting factor will vary depending on the contract.

5.4 State sharding / replication

State sharding and replication are two related types of strategy which might be considered somewhat overlapping and/or the same, so we will look at them both at the same time.

The idea of these strategies is that we can increase throughput for interactions which both consume and produce dapp state stored in UTXOs by spreading the state out across multiple UTXOs.

In my mind, at this moment, the biggest rift in these strategies is between those which use a UTXO-normalized data model and those which use a non-UTXO-normalized data model. I defined a UTXO-normalized data model as one where a transaction does not invalidate any state in any UTXO which is not consumed by that transaction.

In a UTXO-normalized data model, the expected interaction throughput for transactions consuming at least one UTXO in the data model is no more than $O(n)$ for n the number of UTXOs in the data model.

On the other hand, in a case of a non-UTXO-normalized data model, the UTXO consumption of an interaction may include not only the UTXO consumption (if any) for the transaction to effect the interaction, but also any additional UTXO consumption which is then necessary to deal with the fact that a data dependency crossed UTXO boundaries and data in a UTXO not consumed by a transaction was thereby invalidated. What exactly that looks like is hard to say without seeing the model.

Let's look at an example of a UTXO-normalized state sharing / replication model for a DEX, and also a non-UTXO-normalized state sharing / replication model for a DEX.

For the UTXO-normalized model of a DEX, we will use a Uniswap-like DEX concept. [10] In this concept, liquidity providers deposit funds into the DEX contract in order to mint liquidity tokens. The number of liquidity tokens minted when depositing funds, as well as the prices of trades, are determined by solving an equation. When liquidity providers want to be paid back, they send their liquidity tokens to the contract and receive a proportional share of the contract funds including proceeds from trading fees. This share is proportional to the ratio between the liquidity tokens being sent back and the total number of liquidity tokens.

In this DEX concept, as implementable in a UTXO data model, the data required to compute prices is stored in contract state UTXOs. This data includes the amount of each asset which is present in the liquidity pool, and the total number of liquidity tokens minted. Generally, each interaction changes the state, and thus, a single-threaded state machine model makes sense for this DEX concept.

Now, let's look at a concept of a DEX along the lines of Uniswap but which is implemented using the state sharding / replication pattern to support massive interaction throughput.

The basic idea here is to have multiple semi-independent liquidity pools which follow the same contract but have separate state UTXOs. Each of

these pools has potentially different amounts of liquidity of each asset class in the pools. All of the pools of the contract contain the same asset classes. Each of them potentially has a different associated liquidity token supply. They may or may not use the same liquidity token asset classes.

There are a lot of design options to consider in implementing this concept. Let us briefly look at just a few of the considerations.

Using the same liquidity token asset class for each pool means that liquidity token pricing differences between different pools caused by different values in state UTXOs may give rise to arbitrage opportunities. Using different liquidity token asset classes on different pools, on the other hand, may give rise to a less than ideal user experience, in that users will experience holding numerous different liquidity token asset classes, each of which only works on one pool, which would give rise to increased complexity in cryptocurrency portfolio management for end users.

The number of pools available will determine the maximum throughput, as the interaction throughput per block is equal to the number of pools. However, the maximum throughput will not be realized in practice in all cases. It may happen that some state UTXOs are not consumed on a given block and yet many or even most transactions do not get included on the block. This depends on the way in which off-chain code picks pool state UTXOs to use to build the transactions. The UTXO picking algorithm is probably going to involve some element of randomness, but also some element of optimizing for economic efficiency. Some UTXOs may offer a better price than others at a given moment. Then again, if we go for the one offering the best price, then prima facie, it increases the odds that another person will also submit a transaction against that UTXO which will conflict. Thus, choosing the UTXO that offers the best price potentially reduces the chances of the transaction going through.

Under what conditions should the number of pools increased or decreased? Should this happen automatically, or should it require human intervention? How should those decisions be made, and by whom? Should a decentralized governance protocol make those decisions? Should they be made by the contract maintainers? Should they be made by an algorithm programmed into the contract?

Under what conditions should currency be rebalanced (i.e. moved from one pool to another)? How should this happen? Should it happen naturally, as a result of incentives of market participants? Should it happen as a result of processes built into the contract or performed automatically by the con-

tract maintainers? Should it happen as a result of constraints built into the contract?

What pricing algorithm should be used? An order book is too much data to store in one UTXO. Rather than using an order book, this type of DEX contract may determine prices for transactions by solving an equation, known as an invariant equation. Different invariant equations give rise to markets with different characteristics, including slippage and liquidity provider fees.

These are just a few of the things to consider in designing a DEX based on this type of UTXO-normalized data model. Now let's look at what happens when we replace it with a non-UTXO-normalized data model.

Suppose we want to determine the price of a liquidity token for the purposes of depositing liquidity (minting liquidity tokens). Suppose we want to do this by a calculation involving the total number of liquidity tokens, not just in that one pool state UTXO, but over all pool state UTXOs. Then how do we determine that total number? If we store it in a single UTXO, then at each time, each interaction of depositing liquidity will contend for that one UTXO. If we store it in multiple UTXOs, then, each time someone mints or destroys liquidity tokens, each of those UTXOs will be invalidated and have to be re-created. It is this latter case that is a non-UTXO-normalized replication data model, where the total number of liquidity tokens is stored in multiple UTXOs, all of which are invalidated by any change in the total number of liquidity tokens.

That is one example of how to have a DEX with a non-UTXO-normalized data model: have a DEX data model consisting of multiple pool state UTXOs where the same state value (the total liquidity token supply over all state UTXOs) is replicated in each state UTXO, so that if it ever changes, then all state UTXOs are thereby invalidated.

In this example, when we go to the non-UTXO-normalized data model, maximum throughput for adding liquidity, which was $O(n)$ in the UTXO-normalized data model (n the number of UTXOs), drops all the way to $O(1)$. If all existing UTXOs must be invalidated by the process of adding liquidity, then in every block where that is about to happen, none of those UTXOs will be used for anything else. The non-UTXO-normalized model of a DEX here provides parallelism for trading interactions but not for interactions of adding liquidity, and it does not provide for parallelism between interactions of adding liquidity and any other interactions.

5.5 Combination of buffering and state sharding / replication

There is no reason we can't use a combination of buffering and state sharding / replication strategies. The buffering strategy alone leads to an $O(1)$ limiting factor on throughput based on UTXO contention, but it can potentially multiply the number of interactions per transaction by orders of magnitude. In combination with state sharding / replication strategies, buffering strategies may reduce the number of UTXOs required to support a given level of throughput. This may be of importance, for example for the case of a DEX, where the number of pool UTXOs may affect the size (in terms of net asset value) of the UTXOs, and this in turn may affect the pricing / slippage. Since having more pools may adversely affect slippage (if slippage is greater where pools are smaller), using the buffering strategy may help to reduce slippage by reducing the number of pools required to achieve the desired maximum throughput.

6 Potential advantages and issues of known scalability strategies

Consuming no UTXOs avoids all UTXO contention issues but may not always allow for modeling the intended semantics of the contract.

The single threaded state machine pattern may work for you if you do not need high throughput. It has the (potential) advantage of simplicity.

The buffering pattern is not an all purpose solution to achieving scalable throughput, because by itself it does not provide a solution to scalability. However, at the cost of complexity, it may help to optimize other metrics for some projects, like when it's better to use fewer UTXOs to achieve the same amount of throughput. It may be helpful even for projects which require massive scale when it is used in combination with state sharding / replication patterns.

State sharding / replication patterns provide unlimited potential to scale throughput. However, they require us to think carefully about the topology of our data model in terms of its organization into UTXOs. We can choose a UTXO-normalized data model, where data dependencies do not cross UTXO boundaries, which turns out to be a fairly stringent constraint on a state sharding / replication model. Or, we can pay the price of having

a non-UTXO-normalized data model, which is that we need some means of reconciling the temporary inconsistency that results when a transaction invalidates some state stored in a UTXO which it does not consume, ideally without the temporary inconsistency leading to any permanent consequences.

7 Overall recommendations

If you require massive throughput for a Plutus dapp, then look at using the not consuming UTXOs pattern and/or replication / state sharding patterns to achieve the throughput you require. Maybe look at buffering as an optimization add-on to another strategy.

Use asymptotic analysis to predict scalability issues in the design phase. If UTXO contention is a limiting factor, then use asymptotic analysis to estimate how that limiting factor grows with any variable(s) which you can adjust to scale the maximum throughput.

Consider the impact of design changes intended for scalability not just on scalability but on all metrics of interest for the project.

References

[1] Kenny Li. *'Yes, blockchain has a scalability problem. Here's what it is, and here's what people are doing to solve it.'* Hackernoon, 2019.

[2] *'Big O notation.'* Wikipedia.

[3] Fernando Sanchez. *'Cardano's Extended UTXO accounting model – built to support multi-assets and smart contracts.'* IOHK, 2021.

[4] Fernando Sanchez. *'Cardano's Extended UTXO accounting model – built to support multi-assets and smart contracts (part 2).'* IOHK, 2021.

[5] Steve Awodey. *'Category Theory.'* Oxford Logic Guides, 2006. ISBN-13 978-0199237180

[6] Plutus. IOHK, 2021. Commit cc20eb106fc68f81e0e434e125e297c6b083c6ba

[7] *'How to write a scalable Plutus app.'* IOHK, 2021.

[8] Manuel Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, Philip Wadler. *'The Extended UTXO Model.'* IOHK, 2020.

[9] *'The cryptography behind Cardano blocks.'* Cardanians.io (CRDNS pool), 2020.

[10] Hayden Adams. *'Uniswap Whitepaper.'* 2020.