Efficient State Management in Distributed Ledgers

Dimitris Karakostas[1,2], Nikos Karayannidis[2], and Aggelos Kiayias[1,2]

[1] University of Edinburgh
[2] IOHK
dimitris.karakostas@ed.ac.uk,nikos.karagiannidis@iohk.io,akiayias@inf.ed.ac.uk

Abstract. Distributed ledgers implement a storage layer, on top of which a shared state is maintained in a decentralized manner. In UTxO-based ledgers, like Bitcoin, the shared state is the set of all unspent outputs (UTxOs), which serve as inputs to future transactions. The continuously increasing size of this shared state will gradually render its maintenance unaffordable. Our work investigates techniques that minimize the shared state of the distributed ledger, i.e., the in-memory UTxO set. To this end, we follow two directions: a) we propose novel transaction optimization techniques to be followed by wallets, so as to create transactions that reduce the shared state cost and b) propose a novel fee scheme that incentivizes the creation of "state-friendly" transactions. We devise an abstract ledger model, expressed via a series of algebraic operators, and define the transaction optimization problem of minimizing the shared state; we also propose a multi-layered algorithm that approximates the optimal solution to this problem. Finally, we define the necessary conditions such that a ledger's fee scheme incentivizes proper state management and propose a state efficient fee function for Bitcoin.

1 Introduction

The seminal work of Shostak, Pease, and Lamport, during the early '80s, introduced the consensus problem [18,27] and extended our understanding of distributed systems. 30 Years later, Bitcoin [25] introduced what is frequently referred to as "Nakamoto consensus" and the blockchain data structure, followed by widespread research on distributed ledgers.

In ledger systems, participants maintain a shared state which consists of three objects: i) the public ledger, i.e., the list of transactions which form the system's history; ii) the mempool, i.e., the set of, yet unpublished, transactions; iii) the active state which, in systems like Bitcoin, consists of the UTxO set. To support thousands (or millions) of participants, a decentralized system's state management should be well-designed, primarily focused on minimizing the shared state. Our work focuses on the third type, as poorly designed management often leads to performance issues and even Denial-of-Service (DoS) attacks. In Ethereum, during a 2016 DoS attack, an attacker added 18 million accounts to the state, increasing its size by 18 times [35]. Bitcoin saw similar spam attacks in 2013 [32] and 2015 [2], when millions of outputs were added to the UTxO set.

Problem Statement. Mining nodes and full nodes incur costs for maintaining the shared state in the Bitcoin network. This cost pertains to the resources (i.e., CPU, disk, network bandwidth, memory) that are consumed with every transaction transmitted, validated, and stored. An expensive part of a transaction is the newly created outputs, which are added to the in-memory UTxO set. As the system's scale increases, the cost of maintaining the UTxO set gradually leads to a shared-state bloat, which makes the cost of running a full node prohibiting.

Moreover, the system's incentives, which are promoted via transaction fees, only deteriorate the problem. For example, assume two transactions τ_A and τ_B: τ_A spends 5 inputs and creates 1 output, while τ_B spends 1 input and creates 2 outputs. Assuming the size of a UTxO is equal to the size of consuming it (200 bytes) and that transaction fees are 30 satoshi per byte, τ_A costs $30 \times 200 \times (5 + 1) = 36000$ satoshi and τ_B costs $30 \times 200 \times (1 + 2) = 18000$ satoshi. Although τ_B burdens the UTxO set by creating a net delta of $(2 - 1 = 1)$ new UTxO, while τ_A reduces the shared state by consuming $(1 - 5 = -4)$ UTxOs, τ_B is cheaper in terms of fees. Clearly, the existing fee scheme penalizes the consumption of multiple inputs, dis-incentivizing minimizing the shared state.

Our Contributions. Our goal is to devise a set of techniques that minimize the shared state of a distributed ledger, i.e., the in-memory UTxO set. Our approach is twofold: a) we propose transaction optimization techniques which, when employed by wallets, help reduce the shared state's cost; b) propose a novel fee scheme that incentivizes "shared state-friendly" transactions.

In particular, we propose a UTxO model, which abstracts UTxO ledgers and enables evaluating the cost of a ledger's shared state. We then propose a transaction optimization framework, based on three levels of optimization: a) a declarative (rule-based) level, b) a logical/algebraic (cost-based) level, and c) a physical/algorithmic (cost-based) level. Following, we propose three transaction optimization techniques based on the aforementioned optimization levels: a) a rule-driven optimal total order of transactions (the *last-payer rule*), b) a logical transaction transformation (the *2-for-1 transformation*), and c) a novel *input selection* algorithm that minimizes the UTxO set increase, i.e., favors consumption over creation of UTxOs. We then define the transaction optimization problem and propose a 3-step dynamic programming algorithm to approximate the optimal solution. Finally, we define the state efficiency property that a fee function should have, in order to correctly reflect a transaction's shared-state cost, and propose a state efficient fee function for Bitcoin.

Related Work. The problem of unsustainable growth of the UTxO set has concerned developers for years. It has been discussed in community articles [13,15], some [1] offering estimations on the level of inefficiency in Bitcoin. Additionally, research papers [28,8,26,12] have analyzed Bitcoin's and other cryptocurrencies' UTxO sets to gain further insight. Engineering efforts, e.g., in Bitcoin Core's newer releases [22], have also focused on improving performance by reducing the UTxO memory requirements. Various solutions have been proposed to reduce the state of a UTxO ledger, e.g., consolidation of outputs [33] can help reduce the cost of spending multiple small outputs. Alternatively, Utreexo [11], uses cryptographic accumulators to reduce the size of the UTxO set in memory, while BZIP [17] explores lossless compression of the UTxO set.

An important notion in this line of research is the "stateless blockchain" [31]. Such blockchain enables a node to participate in transaction validation without storing the entire state of the blockchain, but only a short commitment to it. Chepurnoy *et al.* [7] employ accumulators and vector commitments to build such blockchain. Concurrently, Boneh *et al.* [5] introduce batching techniques for accumulators in order to build a stateless blockchain with a trustless setup which requires constant amount of storage. We consider an orthogonal problem, i.e., constructing transactions in an incentive-compatible manner that minimizes the state, so these tools can act as building blocks in our proposed techniques.

The role of fees in blockchain systems has also been a topic of interest in recent years. Luu *et al.* [19] explored incentives in Ethereum, focusing on incentivizing miners to correctly verify the validity of scripts run on this "global consensus computer". Möser and Böhme [24] investigate Bitcoin fees empirically and observe that users' behavior depends primarily on the client software, rather than a rational cost estimation. Finally, in an interesting work, Chepurnoy *et al.* [6] propose a fee structure that considers the storage, computation, and network requirements; their core idea is to classify each transaction on one of the three resource types and set its fees accordingly.

2 A UTxO Model

We abstract a distributed ledger as a state machine on which parties act. Specifically, we consider only *payments*, i.e., value transfers between parties; a more elaborate model could take into account arbitrary computations on the ledger's data. We note that our model considers only *fungible* assets.

Initially, we assume a ledger state $\mathcal{S}_{init}$, on which a *transaction* is applied to move the ledger to a new state. Transactions that may be applied on a state are *valid*, following a validation predicate. Each transaction is unique and moves the system to a unique state; with hindsight, we assume that the ledger never transitions to the same state (cf. Definition 5), i.e., valid transactions do not form cycles. Figure 1 provides intuition via a simple ledger model.

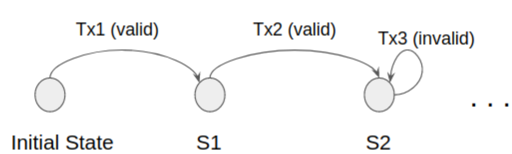

Fig. 1. A decentralized state machine model for a distributed ledger.

Our formalism is similar to chimeric ledgers [36], though focused on UTxO-based ledgers. Following, we provide some basic definitions in a "top-down" approach, starting with the ledger $\mathcal{L}$, which is an ordered list of transactions; our notation of functions is the one typically used in functional programming languages, for example a function $f : A \to B \to C$ takes two input parameters of type A and B respectively and returns a value of type C.

Definition 1. *A* ledger $\mathcal{L}$ *is a list of valid transactions:* $\mathcal{L} \stackrel{def}{=} List[Transaction]$.

A transaction τ transitions the system from one state to another. UTxO-based transactions are thus a product of *inputs*, which define the ownership of assets, and *outputs*, which define the rules of re-transferring the acquired value.

Definition 2. *A UTxO-based transaction* τ *is defined as:* $Transaction \stackrel{def}{=} (inputs : Set[Input], outputs : List[UTxO], forge : Value, fee : Value)$

An *unspent transaction output (UTxO)* represents the ownership of some value from a party, which is represented via an *address* α. Intuitively, in the real world, an output is akin to owning a physical coin of an arbitrary denomination.

Definition 3. *A UTxO is defined as follows:* $UTxO \stackrel{def}{=} (\alpha : Address,\ value : Value,\ created : Timestamp)$.

A transaction's input is a reference to a UTxO, i.e., an output that is owned by the party that creates the transaction. An input consists of two objects: i) the *id* of the transaction that created it (typically its hash) and ii) an index, which identifies the specific output among all UTxOs of the referenced transaction.

Definition 4. *An input is defined as:* $Input \stackrel{def}{=} (id : Hash, index : Int)$.

Given an input and a ledger, three functions retrieve: i) the corresponding output, ii) the corresponding transaction, and iii) the input value. All returned values are wrapped in `Option`, denoting that a value may not be returned.

- $\mathsf{UTxO} : \mathsf{Input} \to \mathcal{L} \to \mathtt{Option}[\mathsf{UTxO}]$
- $\tau : \mathsf{Input} \to \mathcal{L} \to \mathtt{Option}[Transaction]$
- $value : \mathsf{Input} \to \mathcal{L} \to \mathtt{Option}[Value]$

A transaction defines some value that is given as a *fee* to the *miner*, i.e., the party who publishes the transaction into the ledger $\mathcal{L}$. We require that all transactions must preserve value as follows: $\tau.forged + \sum_{i \in \tau.inputs} value(i, \mathcal{L}) = \tau.fee + \sum_{o \in \tau.outputs} o.value$. We note that this applies only on standard transactions, not "coinbase" transactions which create new coins.

Finally, we define the ledger's state $\mathcal{S}$. $\mathcal{S}$ comprises the *UTxO set*, i.e., the set of all outputs of transactions whose value has not been re-transferred and can be used as inputs to new transactions.

Definition 5. *The ledger's state is defined as:* $State \stackrel{def}{=} Set[Input]$.

We now return to the state machine model. A transaction is applied on a ledger state $\mathcal{S}_1$ and results in a ledger state $\mathcal{S}_2$ via the function:

$$\mathsf{txRun} : \mathit{Transaction} \rightarrow \mathit{LedgerState} \rightarrow \mathit{LedgerState}$$

An ordered list of transactions $\mathbb{T} = [\tau_1, \tau_2, \ldots, \tau_N]$ can be applied sequentially on state $\mathcal{S}_1$ to transit to state $\mathcal{S}_N$: $\mathcal{S}_N = (\mathsf{txRun}(\tau_N).\ \ldots\ .\mathsf{txRun}(\tau_2).\mathsf{txRun}(\tau_1))(\mathcal{S}_1)$, assuming the function composition operator ".".

Finally, every ledger state $\mathcal{S}$ corresponds to some cost C. We assume a cost function, which assigns a signed integer of cost units to a ledger state.

$$\mathsf{cost} : \mathit{LedgerState} \rightarrow \mathit{Cost}$$

This function is employed in Definition 6, which defines a transaction's cost; minimizing this cost will be the target of our optimization. Observe that the transaction's cost might be negative, e.g., if the transaction reduces the state.

Definition 6. *The cost of a transaction τ applied to a state $\mathcal{S}$ is the difference between the cost of the final state minus the cost of the initial state:*

$$\mathsf{costTx} : \mathit{Transaction} \rightarrow \mathit{LedgerState} \rightarrow \mathit{Cost}$$
$$\mathsf{costTx}(\tau, \mathcal{S}) = \mathit{cost}(\mathsf{txRun}(\tau, \mathcal{S})) - \mathit{cost}(\mathcal{S})$$

The cost of an ordered list of transactions $[T]$ applied to a state $\mathcal{S}$ is the difference between the cost of the final state minus the cost of the initial state:

$$\mathsf{costTotTx} : [\mathit{Transaction}] \rightarrow \mathit{LedgerState} \rightarrow \mathit{Cost}$$
$$\mathsf{costTotTx}([T], \mathcal{S}) = \mathit{cost}((\mathsf{txRun}(\tau_N).\ \ldots\ .\mathsf{txRun}(\tau_2).\mathsf{txRun}(\tau_1))(\mathcal{S}_1)) - \mathit{cost}(\mathcal{S})$$

We note that, in the rest of the paper, cost represents the size of the ledger's state. However, our model is generic enough to accommodate alternative cost designs as well. For instance, cost could represent the computational effort of producing or verifying the state, such that a cost unit would be a computational cycle. Therefore, our analysis would also be directly applicable in that case, by accordingly adapting some parts of the subsequent optimization framework like the heuristics.

3 Transaction Optimization

The purpose of a distributed ledger is to execute payments, i.e., transfer value from one party to another via transactions. Multiple transactions can perform the same transfer of value between two parties. Such transactions are *equivalent* in terms of their final result, i.e., transferring some value between parties A and B, but may vary in their cost to the ledger state. *Transaction optimization* is the problem of finding the equivalent transaction with minimum cost; our work is heavily inspired by the seminal research on database query optimization [16].

The cost difference between equivalent transactions may be significant. For example, assume that Alice wants to give Bob 100 coins and owns a UTxO of 100 coins and 100 UTxOs of 1 coin each. Consider the two equivalent plans: 1) Alice spends the single UTxO of value 100 and creates 100 outputs of value 1 for Bob; 2) Alice spends the 100 UTxOs of 1 coin value and defines a single UTxO of value 100 to transfer to Bob. The cost of the two approaches exemplifies the ledger state impact that equivalent transactions may have. The first plan increases the ledger's state by 99 UTxOs, while the second decreases it by the same amount.

Following, we use the terms plan and transaction interchangeably, i.e., an alternative plan that achieves the same goal is expressed as an alternative, equivalent transaction. Definition 7 describes transaction equivalency, while Definition 8 defines equivalency between two ordered lists of transactions.

Definition 7. *Transactions τ_1, τ_2 are equivalent (denoted $\tau_1 \equiv \tau_2$) if, when applied to the same state $\mathcal{S}_A$ of a ledger $\mathcal{L}$, they result in states $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, with the same total accumulated value per unique address α:*

$$\forall \alpha \in A \sum_{\substack{i \in \mathcal{S}_1 \\ o_i = \mathsf{UTxO}(i, \mathcal{L}) \\ o_i.address = \alpha}} o_i.value = \sum_{\substack{j \in \mathcal{S}_2 \\ o_j = \mathsf{UTxO}(j, \mathcal{L}) \\ o_j.address = \alpha}} o_j.value$$

where A is the set of all addresses of the parties participating in the ledger system.

Definition 8. *Two different totally ordered sets of the same N transactions $[T_i]$ and $[T_j]$ are equivalent (denoted as $[T_i] \equiv [T_j]$) if, when applied to the same ledger state $\mathcal{S}_A$ of a ledger $\mathcal{L}$, they result in states $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, where the total accumulated value per unique address α is the same in both states:*

$$\forall \alpha \in A \sum_{\substack{i \in \mathcal{S}_1 \\ o_i = \mathsf{UTxO}(i, \mathcal{L}) \\ o_i.address = \alpha}} o_i.value = \sum_{\substack{j \in \mathcal{S}_2 \\ o_j = \mathsf{UTxO}(j, \mathcal{L}) \\ o_j.address = \alpha}} o_j.value$$

where A is the set of addresses of all participants in the distributed ledger system.

Following, we define the basic logical operators for expressing a transaction and explore optimization techniques for compiling the optimal transaction plan.

3.1 Transaction Logical Operators - Ledger State Algebra

First, we introduce some basic *logical* operators, i.e., functions used to form a transaction. The operators are regarded as basic logical steps for executing a transaction, i.e., irrespective of their particular implementation. However, depending on their implementation, each step may correspond to different cost. The operators operate on and produce a state, forming *transactions* which may be equivalent (cf Definition 7). The operators and operands form a *ledger state algebra* and, as the state is a set of UTxOs (cf. Definition 5), all common set operators are applicable. In case of failure, they return the empty state $\varnothing$.

1. **Input Selection** $\sigma_{(P_{id},V)}$: *LedgerState* $\rightarrow$ *LedgerState*
   $\sigma_{(P_{id},V)}$ is a unary operator, which is given as input parameter a pair *(Party id, Value)*. *Party id* is an abstraction of a set of UTxOs, e.g., it could abstract a *wallet* that controls a set of addresses, each owning multiple UTxOs. When applied on a state $\mathcal{S}_i$, $\sigma_{(P_{id},V)}$ produces a new state $\mathcal{S}_f \subset \mathcal{S}_i$, where $\forall o \in \mathcal{S}_f : o \in P_{id}$ and $\sum_{o \in P_{id}} o.value \geq V$. Essentially, σ is a filter over a state, selecting the UTxOs with aggregate value larger than, or equal to the input V.
2. **Output Creation** $\pi_{[(a_1,v_1),\ldots,(a_n,v_n)]}$: *LedgerState* $\rightarrow$ *LedgerState*
   $\pi_{[(a_1,v_1),\ldots,(a_n,v_n)]}$ is a unary operator, which is given a set of *(Address, Value)* pairs and is applied on a state $\mathcal{S}_i$. It produces a new UTxO set $\mathcal{S}_f$ with $\mathcal{S}_f \cap \mathcal{S}_i = \varnothing$, i.e., $\mathcal{S}_f$ includes only new UTxOs. Also $\forall o \in \mathcal{S}_f$: $(o.address, \sum_{o.address} o.value) \in [(a_1, v_1), \ldots a_n, v_n)]$, i.e., the aggregate output value per address is equal to the input parameter. We require that value is preserved, i.e., the total value in $\mathcal{S}_i$ is greater than (or equal to) the total value in $\mathcal{S}_f$; the value difference is is the miners' fee.
3. **Transaction Validation** $\tau_{V_R,\mathcal{S}_i}$: *LedgerState* $\rightarrow$ *LedgerState* $\rightarrow$ *LedgerState*
   $\tau_{V_R,\mathcal{S}_i}$ is a binary operator that validates input and output states $\mathcal{S}_I, \mathcal{S}_O$, against a set of rules V_R, over an initial state $\mathcal{S}_G$. If validation succeeds, it returns an updated state $\mathcal{S}_f = (\mathcal{S}_G - \mathcal{S}_I) \cup \mathcal{S}_O$.

Figure 2 depicts the simplest transaction under our algebra, i.e., a tree with a root and two branches. The root is the transaction validation operator (τ) that receives two inputs: a) the set of selected inputs (σ on the left branch) and b) the set of outputs to be created (π on the right branch). Algebraically we express this transaction as: $T = (\sigma_{Alice,V})'\tau'(\pi_{Bob,V})$, $'\tau'$ being the infix validation operator.

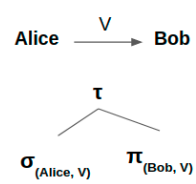

Fig. 2. The simplest expression of a transaction.

Moving one step further, we assume three transactions τ_1, τ_2 and τ_3. The execution of these transactions is *totally ordered*, i.e., $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. Figure 3 depicts this expression. Here, τ_1 is nested within τ_2 and both are nested within τ_3. Such tree is executed from bottom to top, therefore τ_2 is given the ledger state generated after τ_1 is executed; similarly, τ_3 is given the ledger state generated after both τ_1 and τ_2 are executed. Given the above, we next define *subtransactions*; interestingly, transactions may spend outputs created from their subtransactions, thus we also define the notion of *correlated transactions*.

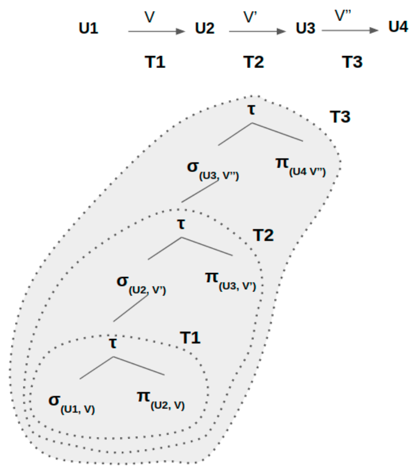

Fig. 3. The expression tree entails a transaction execution total order.

Definition 9. *A* subtransaction *is a transaction nested within a "parent" transaction; it is executed first, so its impact on the ledger state is visible to the parent.*

Definition 10. *Two transactions τ_1, τ_2 are correlated, if τ_1 is a subtransaction of τ_2 and τ_2 spends at least one output created by τ_1.*

3.2 A Transaction Optimization Framework

We now identify different phases in the transaction optimization process; in a hypothetical *transaction optimizer* each phase would be a distinct module. These phases are different approaches to producing equivalent transactions. The phases operate on three levels of optimization: a) a declarative (rule-based) level, b) a logical/algebraic (cost-based) level, and c) a physical/algorithmic (cost-based) level, as depicted in Figure 4. The input of the process is a transaction set $[\tau_x]$, that we want to optimize, and the output is the optimal transaction $\tau_{x-Optimal}$.

Rules: This phase is declarative, as it does not depend on the cost; instead, when applied, it necessarily produces a better transaction. Essentially it consists of *heuristic rules* that are applied by default to produce an equivalent transaction; example of such rules are "create a single output per address" or "consume as many inputs and create as few outputs as possible".

Algebraic Transformations: These are transformations at the level of logical operators that define a transaction's execution. Generally the efficiency of such transformation is evaluated based on the entailed cost. Examples of such transformations are the 2-for-1 transformation (cf. Definition 11) and different transaction orderings (cf. Definition 9).

Methods and Structures: This phase optimizes the algorithm that implements a logical operator. For instance, given two algorithms A, B result in transaction costs C_A, C_B, if $C_A < C_B$ we would choose A; one such example is the different implementations of the input selection operator σ, as shown in Figure 5. Optimizations in this phase may also change the data structure used to access the underlying data, which in our case is the ledger state.

Planning and Searching: This phase employs a *searching strategy* to explore the available space of candidate solutions, i.e., equivalent transaction plans. This space consists of the transactions produced from the above phases, each evaluated based on their cost, under the available cost model.

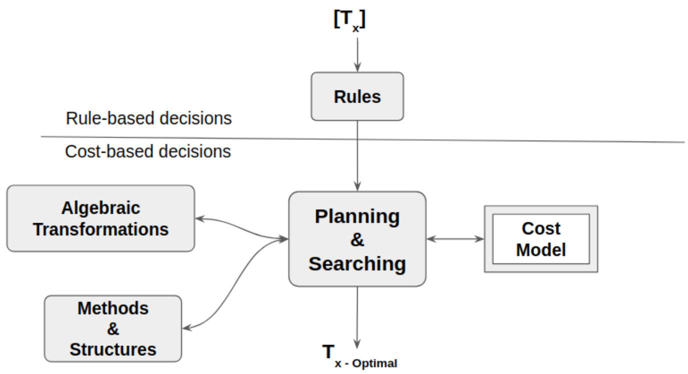

Fig. 4. The transaction optimization process.

3.3 Transaction Optimization Techniques

In this section, we propose three transaction optimization techniques based on the aforementioned optimization levels: a) heuristic rule-based, b) logical/algebraic transformation cost-based, and c) physical/algorithmic cost-based.

Input Selection Optimization We demonstrate this technique with an example. Assume Alice wants to give Bob 5 coins. Figure 5 depicts three equivalent transactions for implementing this payment. Observe that each plan is represented as a tree, where the intermediate nodes are the previously defined logical operators (that act on a ledger state) and the leaf nodes are ledger states. We also assume that the state cost is the number of elements (UTxOs) in the state. The three transactions have the same structure, i.e., they are the same logical expression, but result to different ledger states with different costs. The transactions differ only in the output of the input selection operator ($\sigma_{(Alice,5)}$), a difference which may be attributed to different implementations of the operator; Section Appendix A provides a novel input selection algorithm that minimizes the net delta of created UTxOs; it favors UTxO consumption over creation.

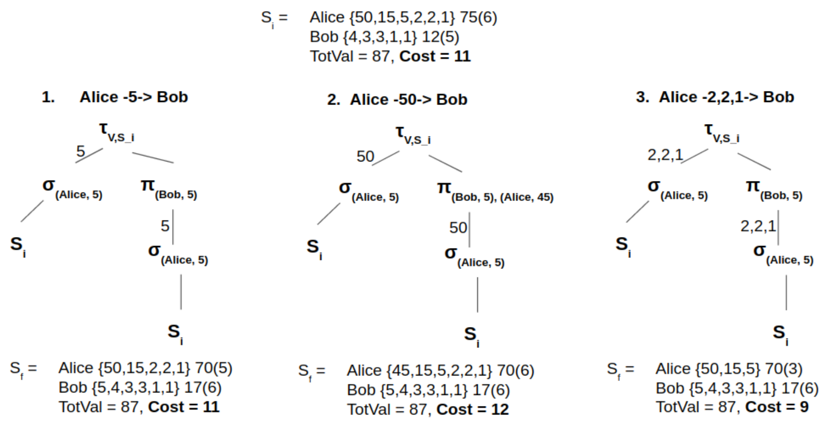

Fig. 5. An example of three equivalent transactions that transfer 5 tokens from Alice to Bob but incur different state costs.

The 2-for-1 Transformation We again consider the example where Alice wants to give Bob 5 coins. Figure 6 depicts a fourth, more complex, equivalent transaction. This transaction consists of two subtransactions (cf. Definition 9), where Alice first gives Bob 17 coins and then receives 12. When the first transaction is completed, an intermediate state (S_i') is created, which is then given as input to the second transaction, that produces the final ledger state $\mathcal{S}_f$ of cost 3. Observe that, although more complex, this transaction minimizes the final ledger state (72% cost reduction). Intuitively, this transaction spends all of Alice's outputs with the first sub-transaction and then does the same for Bob with the second sub-transaction. Therefore, the optimal cost does not depend on input selection (like the 3rd plan of Figure 5), but requires the combination of two transactions that implement a single payment, under a specific amount (12). Definition 11 provides a formal specification of the *2-for-1* logical (algebraic) transformation.

Definition 11. *Given a transaction τ_1, which transfers an amount V from party A to B, the algebraic* 2-for-1 *transformation creates an equivalent transaction τ_2, which consists of (a) a subtransaction, which transfers $V + V_c$ from party A to B and (b) an outer transaction, which transfers V_c from party B to A.*

Figure 7 depicts the 2-for-1 algebraic transformation based on an amount V_c. To implement such a scheme we require an *atomic operation*, where the grouped transactions are executed simultaneously. One method to implement the

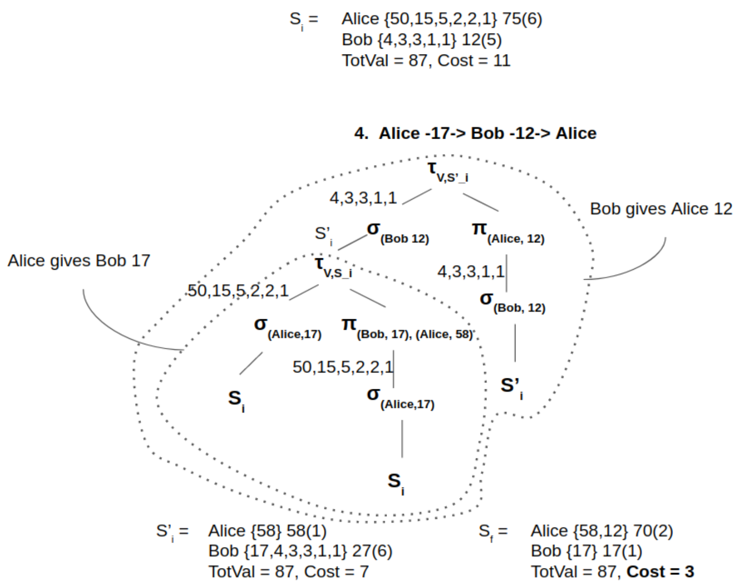

Fig. 6. A 2-for-1 transaction that transfers 5 tokens from Alice to Bob.

atomic transfers is CoinJoin [21], which was proposed for increasing the privacy in Bitcoin; in CoinJoin, the transaction is constructed and signed gradually by each party that contributes its inputs. A similar concept is Algorand's atomic transfers [14], that groups transactions under a common id.

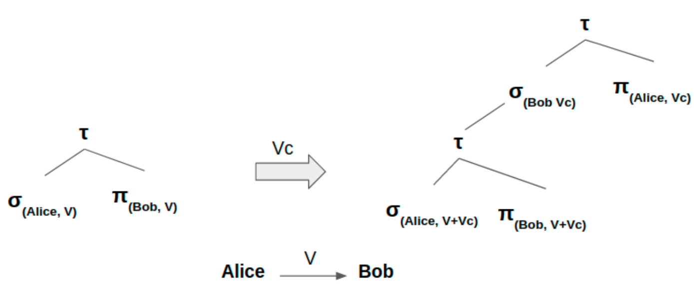

Fig. 7. The 2-for-1 algebraic transformation.

Intuitively, 2-for-1 reduces the transaction's cost by also consuming UTxOs of the receiving party, instead of only consuming outputs of the sending party. Specifically, assume the initial state $\mathcal{S}_i = \{|A|, |B|\}$, where $|A|$ denotes the number of outputs owned by party A. When issuing a payment to B, party A can consume all outputs and consolidate its remaining value to a single UTxO, the "change" output. Such transaction results in state $\mathcal{S}_f = \{1, |B| + 1\}$ with cost $cost(\mathcal{S}_f) = |B| + 2$. If we apply the 2-for-1 transformation, the final state is $\mathcal{S}'_f = \{1+1, 1\}$ with a cost of $cost(\mathcal{S}'_f) = 3$; if $|B| > 1$, then $cost(\mathcal{S}'_f) < cost(\mathcal{S}_f)$. Therefore, if the receiving party has multiple outputs, this transformation creates a transaction with a smaller cost. Consequently, by giving the opportunity to the receiving party of a transaction to spend also its outputs, the 2-for-1 transformation *always* results in a greater shared state cost reduction than the individual un-transformed transaction in the case where there are no fee constraints and thus outputs can be spent freely; otherwise it is a cost-based decision.

Transaction Total Ordering and the Last-Payer Heuristic Rule Assume the following four transactions: (1) T_1: Alice $\xrightarrow{V_1}$ Charlie, (2) T_2: Bob $\xrightarrow{V_2}$ Charlie, (3) T_3: Eve $\xrightarrow{V_3}$ Alice, and (4) T_4: Eve $\xrightarrow{V_4}$ Bob. which are applied on an initial ledger state $\mathcal{S}_i = \{|Alice| = 5, |Bob| = 5, |Charlie| = 2, |Eve| = 3\}$ with cost $cost(\mathcal{S}_i) = 15$; as before, $|A|$ denotes the number of outputs owned by party A and the state cost is the number of all UTxOs.

A first execution order is as follows: $T_1 \to T_2 \to T_3 \to T_4$. For simplicity and without loss of the generality, we assume that when a party pays, it always consumes all available outputs, thus having a single output afterwards (the leftover balance). Similarly, when a party gets paid, the number of UTxOs that it owns increases by one. The state changes with each executed transaction:

i) $\mathcal{S}_i = \{|Alice| = 5, |Bob| = 5, |Charlie| = 2, |Eve| = 3\}, cost = 15$
ii) $T_1 : \{|Alice| = 1, |Bob| = 5, |Charlie| = 3, |Eve| = 3\}, cost = 12$
iii) $T_2 : \{|Alice| = 1, |Bob| = 1, |Charlie| = 4, |Eve| = 3\}, cost = 8$
iv) $T_3 : \{|Alice| = 2, |Bob| = 1, |Charlie| = 4, |Eve| = 1\}, cost = 8$
v) $T_4 : \{|Alice| = 2, |Bob| = 2, |Charlie| = 4, |Eve| = 1\}, cost = 9$

Under a different order, $T_3 \to T_4 \to T_1 \to T_2$, the cost of the final state would be 7. Evidently, the different execution order results in different resulting state cost. Therefore, by changing the nesting order of the transactions in an expression tree, different plans may conduct the same payment with different cost.

Intuitively, parties should have the ability to consume outputs that are produced by the other transactions. For instance, regarding T_1 and T_3, the order $T_3 \to T_1$ is more cost effective ($cost = 10$) than $T_1 \to T_3$ ($cost = 11$), since Alice can consume the output created by Eve. Specifically, if in the last transaction where $\mathcal{P}$ participates, either as a sender or a receiver, $\mathcal{P}$ is the sender, then it can minimize its state cost; we call this the *last-payer heuristic rule*.

Ensuring that each party participates in their last transaction as a sender is not always feasible. Specifically, conflicts may arise in cyclic situations, where $\mathcal{P}_1$ pays $\mathcal{P}_2$ (T_{12}) and also $\mathcal{P}_2$ pays $\mathcal{P}_1$ (T_{21}). Here, it is impossible for both $\mathcal{P}_1$ and $\mathcal{P}_2$ to be the sender in their last transaction. Algorithm 1 below, achieves a transaction ordering based on the last-payer heuristic that bypasses conflicts. This algorithm has a time complexity of $O(M \log M)$ in the number M of participants.

Algorithm 1: Transaction ordering algorithm based on the Last-Payer heuristic rule.

Input: A set of M participants $Set[\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_M]$
Input: A set of k transactions $Set[T_{ij}], i, j = 1, 2, \ldots, M$ among these participants to be ordered. Assume that in transaction T_{ij} party $\mathcal{P}_i$ pays party $\mathcal{P}_j$ ($\mathcal{P}_i \xRightarrow{V_{ij}} \mathcal{P}_j$). Also assume that the transactions are not correlated (see definition 10) and thus all orders are equivalent (see definition 8).
Output: A totally ordered set of transactions $[T_{ij}]$.

1 $output \longleftarrow \emptyset$ $[FinalOrderOfTransactions] \longleftarrow \emptyset$
2 $[OrderedParticipants] \longleftarrow$ Order the input set of participants in an ascending order of the number of received payments.
3 $[ParticipantsLastPaymentAdded] \longleftarrow \emptyset$
4 **while** $[OrderedParticipants] \neq \emptyset$ **do**
5 $\quad \mathcal{P}_{current} \longleftarrow$ get and remove first item from $[OrderedParticipants]$
6 $\quad T_X \longleftarrow$ Find and then remove from $Set[T_{ij}]$, a transaction that $\mathcal{P}_{current}$ pays some participant $\mathcal{P}$ where $\mathcal{P} \notin [ParticipantsLastPaymentAdded]$
7 $\quad$ **if** $T_x == \emptyset$ **then**
8 $\quad\quad$ continue; `/* continue to the next participant */`
9 $\quad$ **else**
10 $\quad\quad [FinalOrderofTransactions] \longleftarrow T_x$; `/* Put it last in the final ordered list */`
11 $\quad\quad [ParticipantsLastPaymentAdded] \longleftarrow \mathcal{P}_{current}$
12 $[FinalOrderOfTransactions] \longleftarrow Set[T_{ij}]$; `/* Add the remaining transactions of the initial set at the beginning (head) of the ordered list */`
13 $ouput \longleftarrow [FinalOrderOfTransactions]$

We provide a short example to demonstrate the inner-workings of Algorithm 1. Assume the four transactions: $T_{12} : \mathcal{P}_1 \rightarrow \mathcal{P}_2$, $T_{21} : \mathcal{P}_2 \rightarrow \mathcal{P}_1$, $T_{13} : \mathcal{P}_1 \rightarrow \mathcal{P}_3$, and $T_{23} : \mathcal{P}_2 \rightarrow \mathcal{P}_3$. First (line 2), the algorithm sorts the list of participants in ascending order of receiving payments, i.e., the more payments a party receives, the more last-payers will conflict, so it should not be considered early-on as a last-payer. In our example, where $\mathcal{P}_3$ receives the most (2) payments, this results in order: $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$. Next (lines 4 - 11), for each party $\mathcal{P}$ in the ordered list, the algorithm tries to find a transaction where $\mathcal{P}$ pays a party who has not been already considered as a last-payer (thus avoiding conflicts); if such transaction exists, it is placed last in the final transaction ordering. Finally, the list of remaining transactions is inserted to the head of the list (line 12). In our example, the transaction ordering through each iteration is: $\mathsf{1st\ iteration} : [T_{12}], \mathsf{2nd\ iteration} : [T_{12}, T_{23}], \mathsf{3rd\ iteration} : [T_{12}, T_{23}], \mathsf{final} : [T_{21}, T_{13}, T_{12}, T_{23}]$. As per the Last-Payer heuristic rule, each party is the sender in their last transaction, except for party $\mathcal{P}_3$ which only receives payments.

Assuming k transactions among M parties, Algorithm 1 is executed locally by each party $\mathcal{P}_i$ after the M participants have coordinated off-chain the k transactions. Specifically, the wallet of each participant exchanges information, in order to gather all k transactions, and then executes the algorithm. The produced total order of transactions will be expressed as a tree of the form depicted in figure 3 and will be implemented as an atomic operation in a similar manner to the 2-for-1 transformation discussed above. Such off-chain coordination for transaction posting is not unique to our work, e.g., this is also how CoinJoin [21] works.

Interestingly, the grouping of many transactions into an atomic operation in general, is a method that can be also aimed at increasing privacy. Therefore, it is an interesting direction for future research to see if it is possible to combine both privacy and space efficiency considerations.

3.4 The Transaction Optimization Problem

Using the above ideas, we now formally define the transaction optimization problem as a typical *optimization problem*, assuming a set of available input selection algorithms $\{Sel_1, Sel_2, \ldots, Sel_l\}$.

Definition 12. *Given N payments between M parties $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_M$ and a search space $\mathcal{S}$ of* equivalent *(cf. Definition 8), ordered lists of transaction plans that execute the N payments, called* candidate solutions*, find the candidate $\tau \in \mathcal{S}$, such that $eval(\tau) \leq eval(\rho)$, for all $\rho \in \mathcal{S}$. Specifically:*

1. *A candidate $\rho \in \mathcal{S}$ is an ordered list of transaction plans*[3] $||T_1|| \rightarrow ||T_2|| \rightarrow \cdots \rightarrow ||T_k||$*, where the transaction plan of a transaction T_x is the pair: $||T_x|| \stackrel{def}{=}$ (Logical Expression, Input Selection Algorithm).*
2. *The search space $\mathcal{S}$ is defined by all candidates $||T_1|| \rightarrow ||T_2|| \rightarrow \cdots \rightarrow ||T_k||$, where, for each transaction T_i, an input selection algorithm is chosen from $\{Sel_1, Sel_2, \ldots, Sel_l\}$ and, possibly, the 2-for-1 logical transformation (cf. Definition 11) is applied.*
3. *eval evaluates the cost of every candidate $\rho \in \mathcal{S}$ (cf. Definition 6) as follows:*

$$\begin{aligned} eval : [Transaction] \rightarrow LedgerState \rightarrow Cost, \\ eval([T_1, T_2, \ldots, T_k], \mathcal{S}_{init}) = \\ cost((txRun(T_k).\ \ldots\ .txRun(T_2).txRun(T_1))(\mathcal{S}_{init})) - cost(\mathcal{S}_{init}) \end{aligned}$$

   where $cost(S) = |S|$ is the size of a ledger state (cf. Definition 5) and $(txRun(T_k).\ \ldots\ .txRun(T_2).txRun(T_1))(\mathcal{S}_{init})$ outputs the final state after the list of transactions is executed on state $\mathcal{S}_{init}$ for each plan $||T_i||$.

Solving the Transaction Optimization Problem We now present a 3-step, dynamic programming algorithm, which solves the transaction optimization problem via an exhaustive search and dynamically pruning candidate solutions:

Step 1: Create N transactions T_{ij}, $i, j \in [1, M]$, corresponding to the N payments $(\mathcal{P}_i \xrightarrow{V_{ij}} \mathcal{P}_j)$, as follows: $T_{ij} = (\sigma_{\mathcal{P}_i, V_{ij}}(\mathcal{S}_{init}))' \tau' (\pi_{\mathcal{P}_j, Vij}(\mathcal{S}_{init}))$ where V_{ij} is the amount to be paid from $\mathcal{P}_i$ to $\mathcal{P}_j$. For each transaction T_{ij}, find the input selection algorithm in $\{Sel_1, Sel_2, \ldots, Sel_l\}$ that minimizes $eval(T_{ij}, \mathcal{S}_{init})$. Then, enforce the *heuristic rule* to create a *single* output

[3] We assume that transactions are non-correlated (cf. Definition 10) and all orderings are equivalent (cf. Definition 8).

per recipient address for each transaction. At the end of this step, the algorithm outputs N transaction plans, i.e., N pairs of transaction's T_{ij} logical expression and the chosen input selection algorithm:

$$||T_{ij}|| = ((\sigma_{\mathcal{P}_i,V_{ij}}(\mathcal{S}_{init}))'\tau'(\pi_{\mathcal{P}_j,Vij}(\mathcal{S}_{init})),\ Sel_s)$$

Step 2: On each transaction plan output of Step 1, perform a 2-for-1 transformation (cf. Definition 11). This step produces a transformed transaction as depicted in Figure 8, based on an amount $p \times V_{ij}$, where p is a configuration parameter of the algorithm, typically in the range $0 < p \leq 1$. Then, for each of the two transactions that comprise the 2-for-1 transformation, choose the input selection algorithm that minimizes the *eval* function and enforce the heuristic rule of a *single* output per recipient address. Finally, accept the 2-for-1 transformed transaction only if its cost (given by *eval*) is smaller than the non-transformed transaction.

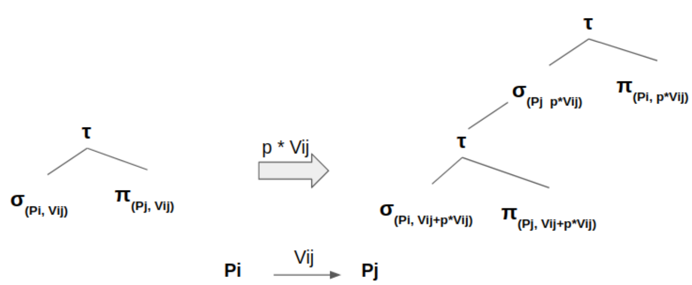

Fig. 8. Applying the 2-for-1 transformation to each separate transaction.

At the end of this step, the algorithm outputs k transaction plans, $k \geq N$, comprising of the 2-for-1 transformed and the non-transformed transactions, along with their input selection algorithms. Importantly, at this point, the algorithm has an optimal plan for each *individual* payment ($\mathcal{P}_i \xrightarrow{V_{ij}} \mathcal{P}_j$), based on an exhaustive search of solutions and cost-driven choices.

Step 3: In this step, the algorithm finds the optimal execution order for the k transactions produced in Step 2. Given the $k!$ permutations, the search space is pruned using the *Last-Payer heuristic rule* (cf. Section 3.3). Finally, the algorithm outputs an ordered list of transaction plans that execute the N payments with a minimum state ledger cost.

As shown, step 2 produces optimal transaction plans, w.r.t. executing the individual transactions, since it performs an exhaustive search for the minimum-cost solution. Step 3 though is based on a heuristic (Last-Payer) to prune the search space, thus only approximating the optimal solution. Future work will evaluate this rule's efficiency and explore techniques to achieve optimality.

4 State Efficiency in Bitcoin

We now define the *state efficiency* property. Our goal is to incentivize users to minimize the global state, without impacting the system's functionality. In that case, if all users are rational, i.e., operate following the incentives, then the state will be minimized as much as possible. Future work will explore the actual impact of deploying such incentives in real-world systems.

To achieve state efficiency, a transaction's fee should be proportional to the incurred state cost. In other words, the more a transaction increases the ledger's state, the higher its fees should be. Specifically, a transaction's fee should reflect: i) the transaction's size, i.e., the cost of storing a transaction permanently on the ledger and ii) the transaction's state cost. A distributed ledger's fee model should aim at incentivizing users to minimizing both storage types, i.e., the distributed ledger and the global state.

First, we define the *fee function* F, i.e., the function that assigns an (integer) fee on a transaction, given a ledger state: $F : \mathit{Transaction} \rightarrow \mathit{LedgerState} \rightarrow \mathit{Int}$. Following, Definition 13 describes *state efficiency.* This property instructs the fee function to (monotonically) increase fees, if a transaction increases the state. Intuitively, between two equivalent transactions, the transaction that incurs greater state cost should also incur a larger fee.

Definition 13. *A fee function F is* state efficient *if*

$$\forall \mathcal{S} \in \mathbb{S} \forall \tau_1, \tau_2 \in \mathbb{T} \mid \tau_1 \equiv \tau_2 \wedge \mathit{costTx}(\tau_1, \mathcal{S}) > \mathit{costTx}(\tau_2, \mathcal{S}) : F(\tau_1, \mathcal{S}) > F(\tau_2, \mathcal{S})$$

for transaction cost function (cf. Definition 6) and equivalence (cf. Definition 7).

Evidently, if the utility of users is to minimize transaction fees, a state efficient fee function ensures that they are also incentivized to minimize the global state. Finally, Definition 14 sets *narrow state efficiency*, a special case of state efficiency which compares equivalent transactions that differ only in their inputs.

Definition 14. *A fee function F is* narrow state efficient *if*

$$\begin{aligned} \forall \mathcal{S} \in \mathbb{S} \forall \tau_1, \tau_2 \in \mathbb{T} \mid \\ \tau_1 \equiv \tau_2 \wedge \tau_1.\mathsf{outputs} = \tau_2.\mathsf{outputs} \wedge \mathit{costTx}(\tau_1, \mathcal{S}) > \mathit{costTx}(\tau_2, \mathcal{S}) : \\ F(\tau_1, \mathcal{S}) > F(\tau_2, \mathcal{S}) \end{aligned}$$

for transaction cost function (cf. Definition 6) and equivalence (cf. Definition 7).

Bitcoin's State Management. Bitcoin's consensus model does not consider fees. Specifically, the user decides a transaction's fees and the miners choose whether to include a transaction in a block. Therefore, it has been stipulated that the level of fees is the balance between the rational choices of miners, who supply the market with block space, and users, who demand part of said space [3].

In practice, most users follow the client software's choice even when not needed [24], e.g., when blocks are not full. Similarly, miners usually follow the hard-coded software rules and may accept even zero-fee transactions. The reference rules of the Bitcoin Wiki [3] define the *fee rate* x, which is the fraction of fees per transaction size, Miners sort transactions based on this metric and solve the Knapsack problem to fill a new block with transactions that maximize it. Some notable alternatives also focus on fee rate [10,30], while reference rules [3] used to also take into account the UTxO age.

As before, a transaction consists of inputs and outputs, i.e., old UTxOs which are spent and newly-created UTxOs. Inputs and UTxOs have a fixed size ι and ω respectively.[4] The size of a transaction is the sum of its inputs and outputs, i.e., is a linear combination of ι and ω, while a transaction's cost is the difference between the number of its UTxOs minus its inputs. Bitcoin's fee function is $F = \beta \cdot \mathsf{size}(\tau)$, where $\mathsf{size}(\tau)$ is τ's size in bytes and β is a fixed fee per byte.[5]

We break the fee efficiency of F via a counterexample. Assume two transactions which are applied on the same ledger state $\mathcal{S}$; for ease of notation, in the rest of the section $F(\tau)$ denotes $F(\tau, \mathcal{S})$. First, τ_1 has 1 input and 1 output, so its state cost is $\mathsf{costTx}(\tau_1, \mathcal{S}) = 0$ and its fee is $F(\tau_1) = \beta \cdot (\iota + \omega)$. Second, τ_2 has 2 inputs and 1 output, i.e., its state cost is $\mathsf{costTx}(\tau_2, \mathcal{S}) = -1$, since it decreases the state; however, its fee is $F(\tau_2) = \beta \cdot (2 \cdot \iota + \omega) = F(\tau_1) + \beta \cdot \iota$. Thus, although $\mathsf{costTx}(\tau_1) > \mathsf{costTx}(\tau_2)$, τ_2's fee is higher, since it is larger.

A better alternative fee function is the following: $F' = \beta \cdot \mathsf{size}(\tau) + \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$. Note that this is state-efficient in our model for a sufficiently small value of β (cf. Section 4.1). Observe with this function, when increasing the UTxO set, a user needs to pay an extra fee ψ per UTxO. Given this change, the reference rules are updated so that, instead of only the fee rate, miners use the scoring function: $\mathsf{score}(\tau) = \frac{\mathsf{fees}(\tau) - \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})}{\mathsf{size}(\tau)}$, where $\mathsf{fees}(\tau)$ are τ's total fees. In market prices, 1 byte of RAM costs $\$3.35 \cdot 10^{-9}$ [23]. The average size of a Bitcoin UTxO is 61 Bytes [9], so a single Bitcoin UTxO costs $\psi = 61 \cdot 3.35 \cdot 10^{-9} = \$2 \cdot 10^{-7}$. Given 10000 full nodes[6], which maintain the ledger and keep the UTxO in memory, the cost becomes $\psi = \$0.002$; equivalently, denominated in Bitcoin[7], the cost of creating a UTxO is $\psi = 22$ satoshi.

This solution incorporates the operational costs of miners, thus it is the rational choice for miners who aim at maximizing their profit. Observe that, after subtracting the fees that relate to UTxO costs, the scoring mechanism behaves the same as the one currently used by Bitcoin miners. Therefore, if users wish to prioritize their transactions, they would again simply increase their transaction's fees; in that case, the UTxO portion of the fees (i.e., $\psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$) remains the same, hence higher fees result in a higher score, similar to the existing mechanism. Also we note that this mechanism is directly enforceable on Bitcoin without the need of a fork.

---

[4] This assumption slightly diverges from the real-world, where UTxOs are typically of varying size depending on the operations in the ScriptPubKey.

[5] $\beta = 0.0067\$$/byte [September 2020] (`https://bitinfocharts.com`)

[6] `https://bitnodes.io` [July 2020]

[7] $1BTC = \$9000$ [July 2020] (`https://coinmarketcap.com`)

4.1 A State efficient Bitcoin.

Intuitively, to make F state efficient we force the creator of a UTxO to subsidize its consumption, i.e., to pay the user who later consumes it. Our fee function is again: $F' = \beta \cdot \mathsf{size}(\tau) + \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$. Assume two transactions τ_1, τ_2 with i_1, i_2 inputs and o_1, o_2 outputs respectively:

$$\mathsf{costTx}(\tau_1) > \mathsf{costTx}(\tau_2) \Leftrightarrow o_1 - i_1 > o_2 - i_2 \Leftrightarrow o_2 - o_1 < i_2 - i_1 \tag{1}$$

F' is state efficient (cf. Definition 13) if:

$$\begin{aligned} F'(\tau_1) > F'(\tau_2) \Rightarrow \\ \mathsf{size}(\tau_1) \cdot \beta + \mathsf{costTx}(\tau_1) \cdot \psi > \mathsf{size}(\tau_2) \cdot \beta + \mathsf{costTx}(\tau_2) \cdot \psi \Rightarrow \\ (i_1 \cdot \iota + o_1 \cdot \omega) \cdot \beta + (o_1 - i_1) \cdot \psi > (i_2 \cdot \iota + o_2 \cdot \omega) \cdot \beta + (o_2 - i_2) \cdot \psi \Rightarrow \\ (o_1 - i_1) \cdot \psi - (o_2 - i_2) \cdot \psi > (i_2 \cdot \iota + o_2 \cdot \omega) \cdot \beta - (i_1 \cdot \iota + o_1 \cdot \omega) \cdot \beta \Rightarrow \\ (i_2 - i_1 + o_1 - o_2) \cdot \psi > ((i_2 - i_1) \cdot \iota + (o_2 - o_1) \cdot \omega) \cdot \beta \overset{(1)}{\Longrightarrow} \\ \psi > \frac{(i_2 - i_1) \cdot \iota + (o_2 - o_1) \cdot \omega}{(i_2 - i_1) - (o_2 - o_1)} \cdot \beta \end{aligned} \tag{2}$$

If F' is narrow state efficient, then $o_1 = o_2$ and the inequality is simplified:

$$\psi > \iota \cdot \beta \tag{3}$$

We turn again to the previous example. For transaction τ_1, with 1 input and 1 output, $F'(\tau_1) = (\iota + \omega) \cdot \beta$ and for transaction τ_2, with 2 inputs and 1 output, $F'(\tau_2) = (2 \cdot \iota + \omega) \cdot \beta - \psi = F'(\tau_1) + \beta \cdot \iota - \psi$. Since Inequalities 2 and 3 ensure that $\psi > \iota \cdot \beta$, the size fee of the extra input in τ_2 is offset by the extra fee ψ, which is paid by the user who creates it. Again to evaluate these variables we consider market prices. The size of a typical, pay-to-script-hash or pay-to-public-key-hash, UTxO is 34 Bytes [4], while the size of consuming it is 146 bytes. Therefore, to make and make present-day Bitcoin (narrow) state efficient, we can set $\omega = 34$, $\iota = 146$, $\beta = 0.0067\$$, and thus $\psi > 0.0978\$$.

However, this approach presents a number of challenges. To enforce F', the fee policy should be incorporated in the consensus protocol and a transaction's validity will depend on its amount of fees. As long as $F'(\tau) > 0$, i.e., a transaction cannot have negative fees, the fee function can be enforced via a *soft* fork. Specifically, this change is backwards compatible, as miners that do not adopt this change will still accept transactions that follow the new fee scheme. However, if $\mathsf{costTx}(\tau) \ll 0$ and possibly $F'(\tau) < 0$, to implement F' we need to establish a "pot" of fees. When a user creates τ with fee $F' = \beta \cdot \mathsf{size}(\tau) + \psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$, the first part ($\beta \cdot \mathsf{size}(\tau)$) is awarded to the miners as before. The second part ($\psi \cdot \mathsf{costTx}(\tau, \mathcal{S})$) is deposited to (or, in case of negative cost, withdrawn from) the pot. In case of negative cost, the transaction defines a special UTxO for receiving the reimbursement. At any point in time, the size of the pot is directly proportional to the UTxO set. Observe that the miners receive the same rewards as before, so their business model is not affected by this change. Finally, the cost of flooding the system with UTxOs increases by ψ per UTxO which, depending on ψ, can render attacks ineffective.

5 Conclusion

Our paper explores optimizations that minimize the state which is shared among the participants of a distributed ledger system. We compose a framework for optimizing transactions on multiple levels, including heuristic rules, algebraic transformations, and alternative sub-routines. Next, we formally define the optimization problem of constructing state efficient transactions and present an algorithm that approximates the optimal solution. Finally, we explore how fees can incentivize proper state management and propose an amended, state efficient fee function for Bitcoin. Our work also proposes various questions. For instance, complex cost models could also consider a UTxO's in-memory lifespan. Furthermore, future work could explore the implications of using a memory hierarchy, instead of storing the entire state in memory.

References

1. Andresen, G.: Utxo uh-oh... (2015), http://gavinandresen.ninja/utxo-uhoh
2. Bitcoin: July 2015 flood attack (2015), https://en.bitcoin.it/wiki/July_2015_flood_attack
3. Bitcoin: Miner fees (2020), https://en.bitcoin.it/wiki/Miner_fees
4. Bitcoin: Protocol documentation (2020), https://en.bitcoin.it/wiki/Protocol_documentation
5. Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. Cryptology ePrint Archive, Report 2018/1188 (2018), https://eprint.iacr.org/2018/1188
6. Chepurnoy, A., Kharin, V., Meshkov, D.: A systematic approach to cryptocurrency fees. In: Zohar, A., Eyal, I., Teague, V., Clark, J., Bracciali, A., Pintore, F., Sala, M. (eds.) FC 2018 Workshops. Lecture Notes in Computer Science, vol. 10958, pp. 19–30. Springer, Heidelberg, Germany, Nieuwpoort, Curaçao (Mar 2, 2019). https://doi.org/10.1007/978-3-662-58820-8_2
7. Chepurnoy, A., Papamanthou, C., Zhang, Y.: Edrax: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968 (2018), https://eprint.iacr.org/2018/968
8. Delgado-Segura, S., Pérez-Solà, C., Navarro-Arribas, G., Herrera-Joancomartí, J.: Analysis of the bitcoin UTXO set. Cryptology ePrint Archive, Report 2017/1095 (2017), https://eprint.iacr.org/2017/1095
9. Delgado-Segura, S., Pérez-Solà, C., Navarro-Arribas, G., Herrera-Joancomartí, J.: Analysis of the bitcoin UTXO set. In: Zohar, A., Eyal, I., Teague, V., Clark, J., Bracciali, A., Pintore, F., Sala, M. (eds.) FC 2018 Workshops. Lecture Notes in Computer Science, vol. 10958, pp. 78–91. Springer, Heidelberg, Germany, Nieuwpoort, Curaçao (Mar 2, 2019). https://doi.org/10.1007/978-3-662-58820-8_6
10. Dos Santos, S., Chukwuocha, C., Kamali, S., Thulasiram, R.K.: An efficient miner strategy for selecting cryptocurrency transactions. In: 2019 IEEE International Conference on Blockchain (Blockchain). pp. 116–123 (2019)
11. Dryja, T.: Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set. Cryptology ePrint Archive, Report 2019/611 (2019), https://eprint.iacr.org/2019/611

12. Easley, D., O'Hara, M., Basu, S.: From mining to markets: The evolution of bitcoin transaction fees. Journal of Financial Economics **134**(1), 91–109 (2019)
13. Frost, E., van Wirdum, A.: Bitcoin's growing utxo problem and how utreexo can help solve it (2019), https://bitcoinmagazine.com/articles/bitcoins-growing-utxo-problem-and-how-utreexo-can-help-solve-it
14. Fustino, R.: Algorand atomic transfers (2019), https://medium.com/algorand/algorand-atomic-transfers-a405376aad44
15. Ichiba Hotchkiss, G.: The 1.x files: The state of stateless ethereum (2019), https://blog.ethereum.org/2019/12/30/eth1x-files-state-of-stateless-ethereum
16. Ioannidis, Y.E.: Query optimization. ACM Comput. Surv. **28**(1), 121–123 (1996). https://doi.org/10.1145/234313.234367, https://doi.org/10.1145/234313.234367
17. Jiang, S., Li, J., Gong, S., Yan, J., Yan, G., Sun, Y., Li, X.: Bzip: A compact data memory system for utxo-based blockchains. In: 2019 IEEE International Conference on Embedded Software and Systems (ICESS). pp. 1–8. IEEE (2019)
18. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3), 382–401 (1982)
19. Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security. pp. 706–719. ACM Press, Denver, CO, USA (Oct 12–16, 2015). https://doi.org/10.1145/2810103.2813659
20. M., E.: An evaluation of coin selection strategies. Master's thesis Karlsruhe Institute of Technology (2016)
21. Maxwell, G.: Coinjoin: Bitcoin privacy for the real world (2013), https://bitcointalk.org/index.php?topic=279249.msg2983902#msg2983902
22. Maxwell, G.: A deep dive into bitcoin core v0.15 (2017), http://diyhpl.us/wiki/transcripts/gmaxwell-2017-08-28-deep-dive-bitcoin-core-v0.15/
23. McCallum, J.C.: Historical memory prices 1957+ (2020), https://en.bitcoin.it/wiki/Miner_fee://jcmit.net/memoryprice.htm
24. Möser, M., Böhme, R.: Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In: Brenner, M., Christin, N., Johnson, B., Rohloff, K. (eds.) FC 2015 Workshops. Lecture Notes in Computer Science, vol. 8976, pp. 19–33. Springer, Heidelberg, Germany, San Juan, Puerto Rico (Jan 30, 2015). https://doi.org/10.1007/978-3-662-48051-9_2
25. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
26. Nicolas, H.: The economics of bitcoin transaction fees. SSRN Electronic Journal (02 2014). https://doi.org/10.2139/ssrn.2400519
27. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2), 228–234 (1980)
28. Pérez-Solà, C., Delgado-Segura, S., Navarro-Arribas, G., Herrera-Joancomart, J.: Another coin bites the dust: An analysis of dust in UTXO based cryptocurrencies. Cryptology ePrint Archive, Report 2018/513 (2018), https://eprint.iacr.org/2018/513
29. Pérez-Solà, C., Delgado-Segura, S., Navarro-Arribas, G., Herrera-Joancomart, J.: Another coin bites the dust: An analysis of dust in utxo based cryptocurrencies. Cryptology ePrint Archive, Report 2018/513 (2018), https://eprint.iacr.org/2018/513
30. Rizun, P.R.: A transaction fee market exists without a block size limit (2015)
31. Todd, P.: Making utxo set growth irrelevant with low-latency delayed txo commitments (2016), https://petertodd.org/2016/delayed-txo-commitments

32. Vasek, M., Thornton, M., Moore, T.: Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014 Workshops. Lecture Notes in Computer Science, vol. 8438, pp. 57–71. Springer, Heidelberg, Germany, Christ Church, Barbados (Mar 7, 2014). `https://doi.org/10.1007/978-3-662-44774-1_5`
33. Wiki, B.: How to cheaply consolidate coins to reduce miner fees (2020), `https://en.bitcoin.it/wiki/How_to_cheaply_consolidate_coins_to_reduce_miner_fees`
34. Wikipedia: Knapsack problem (2020), `https://en.wikipedia.org/wiki/Knapsack_problem`
35. Wilcke, J.: The ethereum network is currently undergoing a dos attack (2016), `https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/`
36. Zahnentferner, J.: Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint Archive, Report 2018/262 (2018), `https://eprint.iacr.org/2018/262`

A A State-friendly Input Selection Algorithm

As shown in Section 3.3, the *input selection*[8] algorithm, which implements the logical operator σ (cf. Section 3.1), directly impacts the state cost of created transactions. Therefore, optimizing the input selection process is a reasonable step towards the optimization of ledger state. Algorithm 2 presented below describes a new input selection algorithm that optimizes transaction cost. In particular, the algorithm's main goal is to minimize the *net amount (delta)* of a new transaction's outputs. Ideally, a transaction should have a *negative delta*, i.e., reduce the ledger state by consuming more outputs than it creates. We again assume that the cost of a ledger state is its size, i.e., the number of UTxOs.

The algorithm receives as input a ledger state, the target value that should be sent to each address, and the number of outputs (excluding change) that we wish our transaction to have. We assume that the outputs of the transaction are fixed and that the optimization is at the level of the input selection. Additionally, we assume that, like Bitcoin, the fees are calculated based on the transaction's size:

$$fees = (|Inputs| + |Outputs| + 1) \cdot utxoSizeBytes \cdot costPerByte$$

Observe that consuming more inputs increases the transaction's size and, consequently, its fee, so the algorithm is also given a maximum allowed fee amount per address. Ultimately, the input selection algorithm aims at selecting as many inputs (i.e., available UTxOs) as possible, such that the funds suffice without exceeding the fee cap. The input selection algorithm outputs a set of UTxOs to be used as inputs of the new transaction.

The algorithm finds the largest set of inputs such that both the fee cap and the spending amount are satisfied. Specifically, first, it sorts the available inputs by value in ascending order (line 12). Second, it identifies the maximum

[8] In the literature this is also sometimes called *coin selection*.

amount of inputs that can be consumed, such that the fee cap is respected (lines 13-22). Third, it adjusts the input selection, so that the selected inputs provide sufficient amount w.r.t. the target value and the corresponding fees (lines 27-33); the selected number of inputs is the same as in the second step. In case funds are insufficient, the algorithm fails with no output returned.

For example, assume that Alice's wallet controls the following UTxOs to be used as inputs: $[10^{-5}, 10^{-5}, 1.0, 2.0, 3.0, 10.0, 50.0, 60.0, 100.0]$ corresponding to an address α. Also assume that she wants to pay a total value of 99 coins across three different recipient addresses. For simplicity we assume that inputs and outputs have the same size and that an extra output is used for change. For different fee caps, the algorithm returns the following input selections. Evidently, a larger fee cap enables more inputs to be selected.

Max Fee	Input Selection
7.0×10^{-4}	$[100.0]$
8.4×10^{-4}	$[50.0, 60.0]$
9.8×10^{-4}	$[10.0, 50.0, 60.0]$
1.12×10^{-3}	$[3.0, 10.0, 50.0, 60.0]$
1.26×10^{-3}	$[2.0, 3.0, 10.0, 50.0, 60.0]$
1.4×10^{-3}	$[1.0, 2.0, 3.0, 10.0, 50.0, 60.0]$
1.54×10^{-3}	$[1.0 \times 10^{-5}, 1.0, 2.0, 3.0, 10.0, 50.0, 60.0]$

Interestingly, Bitcoin's input selection algorithm [20] has a different goal. Specifically, it tries to select inputs that *exactly match* the target value; if no exact match is found, it resorts to a best-match by trying to solve a 0-1 Knapsack problem, which is a well-known NP-hard problem [34] with time complexity of $O(2^n)$, n being the number of available UTxOs. In comparison, our algorithm's complexity is only $O(n \log n)$, due to the sorting step. Therefore, although finding an exact (or best-match) minimizes change, it often creates small valued UTxOs, called *dust*, that "pollute" the ledger state [29].

Algorithm 2: Input Selection algorithm that optimizes UTxO delta and caps transaction fees.

Input: Input Ledger State, i.e., available outputs to be spent.
Input: Desired number of outputs. This is the specified number of outputs that the transaction should have (excluding the output for change) to take into account in the transaction fees calculation
Input: $[(a_1, v_1), \ldots, (a_N, v_N)]$, set of (Address, Value) pairs specifying the lower bound of the value to be spent per address.
Input: $[feeMax_1, \ldots, feeMax_N]$ a maximum fee per address. I.e., the transaction fee is capped.
Output: $[(a_1, [utxo_{11}, \ldots utxo_{1m_1}]), \ldots, (a_N, [utxo_{N1}, \ldots, utxo_{Nm_N}])]$, the set of UTxOs per address to be used as input in the transaction, where the following properties hold:

- $\sum_{j=1}^{m_i} utxo_{ij}.value \geq v_i, i = 1, \ldots N$
- $totaltransactionfee \leq \sum_{i=1}^{N} feeMax_i$
- The number of UTxOs consumed by the transaction is maximum.

```
outputSet ⟵ ∅
foreach input address a_i do
    inputCoins ⟵ getUtxoSet(a_i) ;          // try to spend all UTxOs of this address
    if sumValue(inputCoins) < v_i then
        exitWithError ;                     // not enough money in this address
    if
      sumValue(inputCoins) ≥ v_i + feeCalc(inputCoins) ∧ feeCalc(inputCoins) ≤ feeMax_i
      then
        outputSet ⟵ outputSet ∪ (a_i, inputCoins)
        continue ;                          // we are finished with this address
    else
        inputCoins ⟵ ∅
        availableCoins ⟵ getUtxoSet(a_i)
        order(avaialbleCoins, ASC) ;        // order by value ascending
        while feeCalc(inputCoins) ≤ feeMax_i do
            if availableCoins = ∅ then
                /* no more coins to select */
                break
            else
                c ⟵ removeSmallestCoin(availableCoins)
                if feeCalc(inputCoins ∪ c) ≤ feeMax_i then
                    inputCoins ⟵ inputCoins ∪ c
                else
                    /* We have reached the fee threshold and don't need c */
                    availableCoins ⟵ addSmallestCoin(c, availableCoins)
                    break
        /* We have chosen the maximum number of coins based on the fee threshold */
        /* Check sufficiency of value */
        if sumValue(inputCoins) ≥ v_i + feeCalc(inputCoins) then
            outputSet ⟵ outputSet ∪ (a_i, inputCoins)
            continue ;                      // we are finished with this address
        else
            /* replace the smallest selected coin with the smallest from the
               available coins so that the number of selected coins remains constant
               */
            while sumValue(inputCoins) < v_i + feeCalc(inputCoins) do
                if availableCoins = ∅ then
                    exitWithError ;   /* Cannot gather enough money under this fee
                      constraint with these available coins */
                else
                    c ⟵ removeSmallestCoin(availableCoins)
                    removeSmallestCoin(inputCoins)
                    inputCoins ⟵ inputCoins ∪ c
            outputSet ⟵ outputSet ∪ (a_i, inputCoins)
            continue ;                      // we are finished with this address
```