Introducing

Bitcoin 9000

A dilettante's guide to Bitcoin scalability.

BIP-9000

(self-assigned)

Quote

"It's kind of fun to do the impossible."
Walt Disney

Goal

Safely scale Bitcoin to process over 9000 transactions.

Abstract

We propose a strategy to scale Bitcoin to a far greater throughput and performance than available today while keeping the risk of centralization and costs to a minimum.

To achieve this we decrease block validation latency with diff blocks, parallelize transaction validation, enable UTXO sharding with transaction input block height annotations, and deploy a series of extension blocks for sustainable capacity increases.

Introduction

Bitcoin is hard to scale because users must validate all transactions in order to validate their own payments. Mining nodes must also validate all transactions and do so with a very low latency in order to avoid losing money on stale blocks. Increasing transaction throughput from non-validating users imposes ever growing validation costs on the network nodes. This increases the risk of network centralization where only a handful of nodes are able to validate the entire chain of transactions. On the other hand, keeping a relatively small hard limit (e.g. 1 Mb block size limit) reduces utility of Bitcoin and requires many users to resort to less secure systems for use cases outside the secure cash storage and large-value payments.

In the present paper we propose a scalability plan consisting of four stages to address these problems.

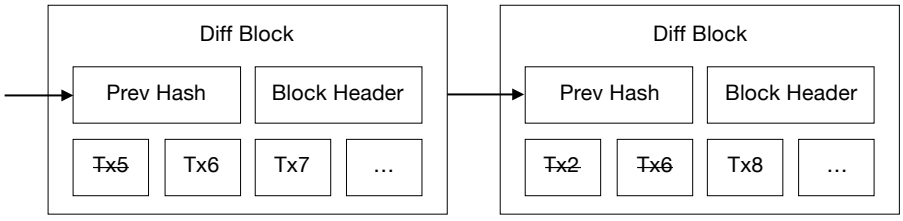
1) Reduce block propagation latency using *diff blocks*. As a side effect, transactions get a better confirmation feedback.
2) Scale transaction validation over *multiple CPUs* to improve throughput and reduce validation latency.
3) Increase capacity using a *series of extension blocks* deployed as a soft fork allowing users to safely boycott unsustainable capacity increases.
4) Assist caching of unspent transaction outputs with *block height annotations* in transaction witness data.

In addition to these we also propose several auxiliary upgrades to improve and extend Bitcoin in the context of increased capacity.

Part 1
Fast Block Distribution Using Diff Blocks

Whenever a new block is discovered by a mining node, it must be delivered and validated by other miners as quickly as possible, so they can begin building their own blocks on top of it. Nodes normally share roughly the same set of transactions in-between blocks, but cannot effectively synchronize on that set before the next block is found.

We observe that mining nodes frequently produce blocks of reduced difficulty that are otherwise valid (sometimes called "weak blocks"). Based on that idea we propose a concept of "diff blocks". Diff blocks are annotated weak blocks with proof-of-work greater or equal to 1/100 of the current target and therefore being produced every 6 seconds on average.
Diff blocks form a tree like the regular blockchain. Unlike blocks in the main chain, every diff block contains the difference in transactions compared to the previous diff block (a list of removed transaction hashes and a list of added transactions). All block headers in the chain of diff blocks have the same *previous block hash* equal to the current tip of the main chain.



Mining nodes are much less concerned with the stale rate of their diff blocks because they do not risk losing their reward. A valid full block may appear on any branch of the diff chain and still will be valid. However, in order to minimize the size of the diff for the valid block, miners strive to build on the existing chain of diff blocks and use cumulative proof-of-work as means to synchronize on the largest possible subset of transactions.

Network

The steps to run the network of diff blocks are as follows:

1) New transactions are broadcast to all mining nodes.
2) Each node collects new transactions in a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work above the diff block threshold, it computes the difference with the state corresponding to the latest known diff block, wraps the discovered block header in a diff block, and broadcasts the diff block to all nodes.
5) Nodes accept the diff block only if all transactions in it are valid and not already spent. Resulting transaction set must match the merkle tree hash in the block header.
6) Nodes express their acceptance of the diff block by working on creating the next diff block in the chain, using the hash of the accepted diff block as the previous hash.
7) If any diff block has proof-of-work matching the full block target, nodes are able to reconstruct the full block from diff data and switch to mining blocks on top of it. Diff blocks not built on top of the new full block are rejected.

Optimization

Since diff blocks appear 100 times more frequently than full blocks, chances of producing stale diff blocks increase. While not a problem for a revenue directly, frequent reorganizations of diff blocks increase validation costs for mining nodes, thus affecting the stale rate of actual blocks. To minimize the costs of chain reorganization, nodes can cache validated transactions.

In case the difficulty target turns out to be too low for the given connectivity latency between mining nodes, they are free to soft fork a higher difficulty (e.g. target/50 yielding diff blocks every 12 seconds) in order to minimize the amount of reorganizations. Alternatively, if the difficulty target is still higher than the existing latency permits, nodes may upgrade to a different version of the message protocol with a lower difficulty threshold for the diff blocks (e.g. target/200 with a 3-second diff block interval).

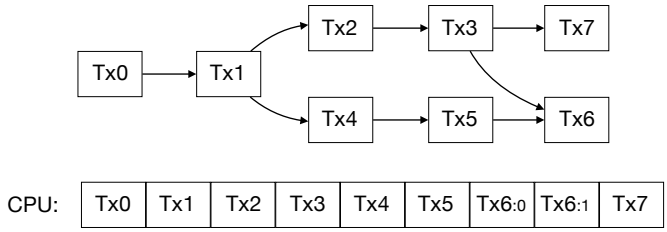
Properties

Diff blocks have several interesting properties.

1) Nodes can see the shared unconfirmed transaction set without keeping track of individual peer's state. This allows to spread the cost of block validation across a 10-minute interval between blocks instead of trying to pack it into a few seconds once the next block is found.
2) Nodes can still replace transactions before the real block is found (features of RBF such as cut-through payments are preserved).
3) No need for canonical transaction ordering, so diff blocks do not conflict with any other protocols that use ordering of transactions today or in the future. If transaction needs to be reordered, diff block should include a deletion entry and re-add transaction in correct position later.
4) Users can get information about the chances their transaction getting mined. Transactions with insufficient fees will not appear in a diff block, which will provide feedback to the user within 6-12 seconds instead of making them wait 10-20 minutes. Note that inclusion in a diff block does not mean that the transaction will be ultimately mined as it can be replaced by another transaction before the full block is found.

Part 2
Scalable Transaction Validation

Once block propagation latency is minimized, a much larger transaction throughput becomes possible. This may shift performance bottleneck to the CPU. Existing Bitcoin implementations perform serial transaction validation. This means that growing capacity of the network increases validation latency linearly constrained by a single core of a modern 2-3 GHz CPU.



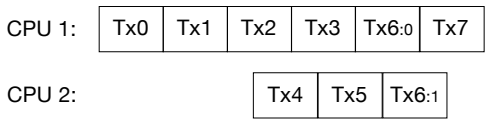
We propose replacing this *latency* requirement with a *bandwidth* requirement by making transaction validation parallelized. This way a growing rate of transactions can be satisfied with a larger array of CPU cores in order to avoid latency increase.

Incoming transactions can be checked for well-formedness in parallel. The only shared data needed is time in the current context (that is, system time in case of memory pool, block timestamp for transactions received in a block) to filter out invalid time-locked transactions.

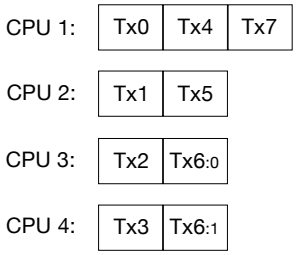
Then, transactions must be verified against double-spending: each input's outpoint must match an unspent output in UTXO set in current context and mark it as spent. This verification can be parallelized by splitting all outpoints in N threads matching N CPUs. Splitting is done by hashing the entire outpoint with node's secret seed. Hashing this way allows nodes to balance the CPU load and prevent potential DoS attacks seeking to cause an imbalanced load on the CPUs.

$$\mathit{ThreadIndex} = (\mathbf{H}(\mathit{Seed} \,||\, \mathit{TxHash}) + \mathit{OutputIndex}) \bmod N$$

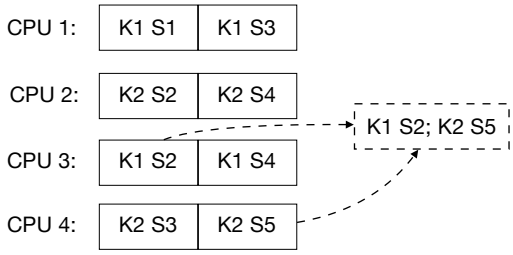
After a successful match with the unspent output, the same thread can be used to execute and verify signature scripts for the given transaction.



Strictly speaking, given a chain of transactions, their signature scripts can be evaluated in parallel, without waiting for the previous transaction to be fully verified. Unfortunately, this opens a possibility of a DoS attack when the attacker submits a long chain of invalid transactions and makes the node fill up its CPUs with useless computations at no cost to the attacker. However, transactions received as part of the block (full or diff) already constitute a sacrifice on the part of a potential attacker. This sacrifice limits DoS surface and permits the node to "look ahead" and validate a limited number of inputs of a not-yet-validated transaction in order to fill otherwise idle CPUs.



Multi-signature verification is also parallelizable, although tricky in some cases. *N*-of-*N* parallelization is obvious (and becomes irrelevant when Schnorr signatures are adopted), but more complex schemes require a trade-off between latency and CPU cost to test multiple combinations in parallel. For instance, 2-of-3 may use up to 4 parallel signature checks while at most 2 will succeed (100% overhead), but 2-of-5 scheme may use up to 8 parallel signature checks (300% overhead). For a larger number of combinations, a "look ahead" window can be used to speculatively check a limited number of key–signature pairs to keep CPU overhead limited. Mining nodes may choose higher CPU overhead over latency (they already spend much more energy on actual mining, so a few extra CPUs are relatively cheap), while non-mining nodes may prefer slightly higher latency and lower CPU consumption.



Optimizing Mining Nodes

While non-mining nodes need enough CPU resources to validate full blocks every 10 minutes, mining nodes might want to go further and reduce the remaining latency more aggressively by parallelizing transaction validation across multiple physical machines. The goal is to perform a minimal number of computations serially. In this case, hash function **H** can be used to break down UTXO set into $M \times N$ groups where M supergroups include whole transactions and allow sharding UTXO set across multiple machines, but N subgroups have individual outputs spread across threads within a single machine.

$$MachineIndex = \mathbf{H}(SeedA \,||\, TxHash) \bmod M$$
$$ThreadIndex = (\mathbf{H}(SeedB \,||\, TxHash) + OutputIndex) \bmod N$$

Assuming the time needed to verify a single transaction input is 1 ms, a 2-core personal computer would be capable of validating 1000 double-input transactions per second. The same throughput on a 10-machine setup with a faster 12-core CPUs yields 17 ms latency for validating the same 1000 transactions appearing each second.

Part 3
Multi-Level Capacity Upgrades

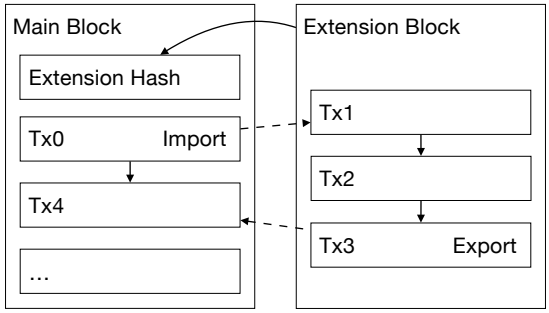
In parts 1 and 2 we discussed block propagation and transaction validation latency improvements. To actually take advantage of these, we need a scheme to continuously increase the capacity of the network in a safe manner.

Many existing proposals to increase upper limit on block size ultimately leave the decision on block size limit to the mining nodes. Miners are always able to *reduce* the capacity: either individually by abstaining from including extra transactions, or collectively by enforcing a limit via a soft fork. All known proposals to allow anyone to vote on *increasing* capacity are reducible to voting performed by miners alone (because they alone decide which and how many transactions get included in a block).

What is needed is the ability for miners *to collectively increase the capacity* while effectively allowing users *to reduce it* when necessary. Miners are able to perform a soft fork upgrade to a higher capacity, while non-miners may choose to ignore it and simply require a relatively high confirmation count for the transactions that they do not validate.

Extension Blocks

We start with the idea of "extension blocks" proposed by Adam Back and deployable via a soft fork. Transactions in a normal block can commit funds to be spent within the extension block. The extension block itself is then committed to the main block. This enables us to place whole chains of transactions within the extension block without consuming the capacity of the main block.



To make a transaction spendable in the extension block, funds must be sent to a new kind of output that we call an *import commitment*. Such commitment specifies the script necessary to unlock the output in the extension block.

Import-committed output can be spent by a transaction in the extension block of the corresponding level within the same main block or later, provided its witness script satisfies the output script and the witness contains the correct block level (0 for main block) that helps locate the output. Then, usual rules apply to transactions within the extension block.

When the funds need to be exported to the main block, an *export commitment* is used that similarly specifies the output script to be used in the main block *export transaction*.

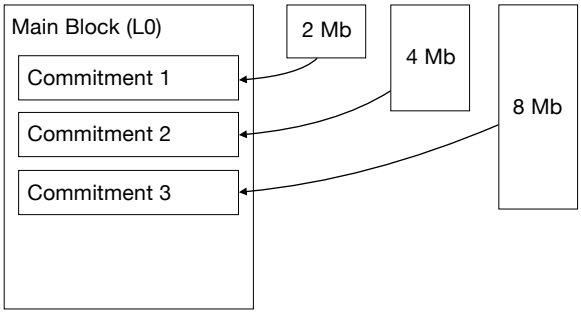
All export-committed outputs in the extension block must be matched with a single export transaction constructed by the miner of the block. Miner is free to choose any subset of unspent import-committed outputs in the main blocks, but must spend them exactly to outputs matching the export commitments. Excess value must be directed to a special change output with a script containing a *change commitment* (similar to import commitment, but without the ability to be spent in the extension block). Change-committed outputs can be spent by the miner alongside the import-committed outputs in order to service exports from the extension block.

This way, an exported transaction output in the extension block is immediately matched (within the enclosing main block) by a corresponding output in the main block and is ready to be spent by its owner.

Multi-Level Extension Blocks

Instead of using just one extension block, we propose a series of extension blocks with exponentially growing capacities of 2 Mb, 4 Mb and so on up to 8192 Mb unlocked one after another via individual soft forks. This sequence would yield a total capacity of over 9000 Mb as required.

Individual transactions declare the level at which they can be included. Mining nodes can collectively vote on unlocking the next level via a soft fork while non-mining nodes may decide whether to use and trust transactions within this new level. Each soft fork also provides an opportunity to adjust the format of the block to introduce improvements and new features to both blocks and transactions.



Higher-order extension blocks are committed recursively into their enclosing blocks. Transaction imports and exports are also recursively applied between levels *N* and *N*+1. This way, multiple hops are needed to move funds across several levels.

Specification Overview

1) In order to prepare all nodes for future upgrades we first need to enforce valid commitments to all extension blocks. The top-level block should include a version bit to indicate support for the extension blocks. Support for extension blocks is enforced when 950 out of 1000 blocks in a row declare such support. This does not yet unlock any specific capacity, but enforces validation of individual commitments to extension blocks for future upgrades.
2) Unlocking an additional extension block level requires another soft fork upgrade. The top-level block should include a version bit to indicate support for the upgrade to the next level extension block. Additional extension block is enforced when 950 out of 1000 blocks in a row declare such support (provided the previous level is already activated).
3) Each extension block has a header containing:
   - Merkle root of included transactions.
   - UTXO commitment for transactions at a given level (see Part 5 for details).
   - Variable-length freeform field for future soft fork extensions and additional commitments for this block level.
4) Every individual extension block is committed via a designated zero-value prunable output in the coinbase transaction of the top-level block with the following script: `OP_RETURN <level> <commitment>`. Already activated and enforced extension blocks are committed by placing their hash in the corresponding output script. Extension blocks in the process of soft fork pre-announcement are committed with a valid level number and an all-zero commitment.
5) Commitment is a variable-length field extensible with future soft fork upgrades. First 32 bytes must contain a hash of a valid extension block header. The hash must be zeroed if the extension block at this level is absent. Next 32 bytes are allocated for proof-of-execution specified below.
6) Diff blocks store all transactions across all extension levels. Every entry has a block level number to indicate the extension block affected by the entry.

7) *Import commitment* script consists of a single-byte "import" flag (`0xe0`) and the remaining bytes representing a versioned script to be spent in the next-level extension block. Such output is added to the UTXO set at the current level (spendable by export transactions only) and the UTXO set at the next level (spendable via the specified script in the next level extension block).
8) *Change commitment* script consists of a single-byte "import change" flag (`0xe1`). This affects only the UTXO set at the current level (spendable by exports only). This script is used to allocate excess funds not exported in the current block.
9) *Export commitment* script consists of a single-byte "export" flag (`0xe2`) followed by a versioned script. This commitment prevents the output from entering the UTXO set at a given level and requires a higher level block to include a transaction spending previous *import-committed* or *change-committed* outputs and allocating the specified value to a specified versioned script.

As an additional precaution for nodes that do not validate transactions at a certain level, export outputs in blocks of all levels are not spendable until reaching maturity of 6 confirmations. Change outputs can be spent without maturity requirement because they are already required to be spent by export transactions only. Without the additional maturity limit on export transactions, miners would have a lower cost opportunity to attack the nodes that do not validate higher-level extension blocks.

Improved Security In Face Of SPV Mining

Both mining and non-mining nodes may find it not cost effective to validate additional capacity levels. This will lead to a "security downgrade to SPV" regarding transactions included in these levels. To minimize the risk of attack by dishonest nodes, users may require an additional confirmation count for export transactions found in the chain leading to their payment. For instance, if a user normally requires 6 confirmations, they may require 12 confirmations for any non-validated export transaction in the ancestor chain. If the latest transaction of that kind is already confirmed by 3 blocks, the total amount of confirmations required for the payment is $max(6, 12 - 3) = 9$.

Mining nodes are also capable of so-called "SPV mining". In this case, an extra capacity may not be efficient to always validate by some nodes that may ignore it and trust the miner of the parent block to validate it correctly. The problem with "SPV mining" is not the individual miner's risk that they take, but that non-mining SPV nodes trust such miner to perform validation and therefore have their risk multiplied if the miner does not actually validate the transactions. To avoid multiplication of risk we propose a "proof-of-execution" (PoE) scheme that indicates if a miner was able to perform full validation of a given block level:

1) The miner chooses whether to verify transactions in a given level *N* or trust ancestor blocks and not include such transactions themselves.
2) If the miner decides to "SPV mine" in respect to level *N*, they do not use a proof-of-execution commitment for that level and are forbidden from including transactions of that level and higher in their block. PoE then should be an all-zero 32-byte string.
3) If the miner decides to fully validate level *N* transactions of the parent block, they must compute and commit to the unique hash of the computation of all transaction inputs by hashing all executed opcodes, values pushed on stack, and relevant signature hashes for `CHECKSIG/CHECKMULTISIG` operations. This commitment does not really prove that the miner actually verified all ECDSA signatures, but proves that all the necessary data was received and processed. This allows to demonstrate to other nodes whether the bandwidth and synchronization are the bottleneck or not for the miners. (We intentionally do not require committing any intermediate data relevant to signature verification to leave room for optimizations and addition of the new signature schemes.)
4) Hashing of each piece of data is performed using HMAC-SHA256 where the key is set to Hash256 image of the coinbase transaction with input signature script blanked out and all zero-value outputs removed. This ensures that PoE is unique to the miner and bound to

the miner's reward without interfering with any other commitment schemes today or in the future.

5) Hashing of transaction inputs and transaction itself is organized in a merkle tree to assist parallelization methods described in Part 2.
6) If the parent block does not include level *N* transactions, PoE applies to the latest block that does.
7) Nodes that do not perform validation of level *N* transactions should discount confirmations of the non-validated transactions by the blocks lacking proof-of-execution for the given transaction's level.
8) All nodes can use PoE to estimate the effect of bandwidth constraints on miners. If a dangerously large amount of miners are performing SPV mining for level *N*, then such level must be considered unreliable and its usage should be discouraged until more miners can afford validating it.

Note that proof-of-execution applies only to the previous block, not to all blocks or a range of blocks. Increasing the range would require hashing the same transactions multiple times which is undesirable. However, it is not a weakness of the scheme. If the miner is committed to process level *N* transactions, they will continuously calculate PoE for all blocks even if they only mine a block once a day. If the miner chooses to process such transactions only X% of the time, it would be equivalent to having only X% of its hashing power using PoE which will be reflected in actual frequency of PoE in mined blocks and act as a useful indication of how much hashing power is performing validation of a given capacity level.

Part 4
Block Height Input Annotations

Larger capacity also means a faster growing set of unspent transaction outputs ("UTXO set"). We need fast access to this set to enable quick transaction validation, which means more expensive RAM storage. However, large amounts of RAM needed to store everyone's unspent coins may become prohibitively costly. Fortunately, some coins are less likely to be spent than others. If we can predict which coins are less likely to be spent we can offload them to a much cheaper, although a slower storage (such as an SSD or even a spinning hard drive).

Taking cues from the design of generational garbage collectors, we propose a *block height* as a heuristic to determine how likely any given output is to be spent. Younger outputs are spent more often while older outputs are less likely to be spent in the next block. This is similar to an observation that "most objects die young" in a typical GC system.

Specific segregation of the UTXO set by the block height is not consensus-critical and becomes an implementation detail. We expect that most nodes will decide to split the UTXO set in two parts: one filling the whole RAM and the other sitting on an SSD.

A new consensus rule is needed to assist with locating the outputs. All blocks are required to annotate transaction inputs with a valid block height as a full 32-bit little-endian integer. This allows nodes to immediately identify the location of the relevant unspent output without additional in-memory index that would defeat the purpose of caching only a part of the UTXO set in RAM.

Block height annotations must necessarily be a part of the witness data since they are added by the miners (miners decide in which block any given transaction is included). This way block heights are not part of the normalized transaction hash.

Part 5
Additional Enhancements

Native Segregated Witness

Separate extension blocks provide an opportunity to modernize the transaction format. Witness data does not need to be committed using a separate tree, but can be committed directly as a merkle tree of *witness hashes* while *outpoint* structures continue to use a *normalized transaction hash* that does not include signature scripts and additional witness data. Such change would be breaking for the main blocks, but can be supported natively in the extension blocks from the start.

Multi-asset transactions

Increased capacity enables many more interesting uses of Bitcoin. In order to continue ещ згыр latte-buying hipsters out of the network, we propose to extend the transaction format to support issuance and transfer of arbitrary assets in addition to the native *bitcoin* currency. This would enable many sorts of applications of smart property and financial assets to participate on the most secure financial network at a negligible cost to the nodes (transaction fees will still be payable in bitcoin).

Assets are defined by their *issuance script* that must be spent to introduce a newly minted asset. Issuance script is identified by a separate script version ("issuance script version" `0xa0`) extended with a little-endian uint64 amount (`PUSHDATA <VersionByte Amount RestOfTheScript…>`). During transaction validation, an input that spends such output is assigned the corresponding amount of that asset (in addition to the amount of bitcoin value assigned to it).

Extension block transactions specify a variable-length *asset identifier* equal to Hash256 of the corresponding issuance script with the amount zeroed out (so that all issuances of different value have the same asset identifier). Native *bitcoin* outputs carry a zero-length asset identifier.

It is open for discussion whether to enforce a strict balance between inputs and outputs of non-bitcoin values, or to allow outputs to have lower value and leave the difference to the mining nodes as a fee.

State Commitments

Increased capacity makes bootstrapping a node a much more difficult task. Scanning the entire historical chain of blocks may become highly impractical. To assist with bootstrapping, every block must commit to a merkle root of all unspent outputs consisting of the *amount*, *asset identifier*, *script*, *block height*, *transaction hash* and *output index*. Outputs are indexed by their *outpoint* (transaction hash + output index, encoded in big-endian to assist sorting). Each extension block has its own commitment for the unspent outputs created at its level.

To support efficient updates to the UTXO commitments we need to use a merkle radix tree (known as "patricia trie") that allows O(log(*N*)) updates. Such construction also allows efficient partial validation enabling lightweight nodes to store and validate smaller random portions of the whole UTXO set and assist the network of SPV nodes with fraud proofs.

Acknowledgements

Presented plan is based on ideas proposed by Gregory Maxwell, Adam Back, Peter Todd, Pieter Wuille, Matt Corallo, Gavin Andresen and other Bitcoin developers.