Merklix tree for Bitcoin

By Joannes Vermorel (Lokad), Amaury Séchet (Bitcoin ABC), Shammah Chancellor (Bitcoin ABC), Jason Cox (Bitcoin ABC), July 2018

In the present document, Bitcoin always refers to Bitcoin Cash

Abstract: The layout of the Merkle tree as originally found in Bitcoin does not lend itself to incremental embarrassingly parallel processing, which will become a hindrance if the blocks are to become large in the future. Thus, we propose the Merklix tree, a change of the Bitcoin consensus rule that impacts the hashing strategy for a block. The Merklix tree is a bitwise trie[1] supplemented by a Merkle overlay. The Merklix tree lends itself to incremental embarrassingly parallel calculations, and it also maximally preserves the backward compatibility with the existing applicative landscape of Bitcoin.

This proposal should be seen as the first part of a two-part proposal for Bitcoin. The second proposal "Metadata subtree for Bitcoin" focuses on the introduction of new block-level metadata, and covers the specifics of the coinbase in greater details. Keeping those two proposals together simplifies the upgrade path for the Bitcoin ecosystem at large as both impact the block format.

The original design of Satoshi's client for Bitcoin organizes the transactions within a block as a Merkle tree. The Merkle tree approach is of high interest because it offers the possibility to produce a variety of compact proofs about transactions without having to cope with entire blocks. Also, it is possible to arrange the Merkle tree to make it suitable for incremental embarrassingly parallel processing. This property is highly desirable in order to scale Bitcoin up to large blocks, potentially reaching 1TB per block.

However, the original implementation of Bitcoin left much to be desired as far as the *layout* of its Merkle tree is concerned. This strategy, referred to as the *balanced Merkle tree* in the following, is not amenable to an incremental embarrassingly parallel implementation that supports concurrent insertions. With significant complications, insertions in the balanced Merkle tree can be parallelized. However, this level of complexity is undesirable for infrastructure software.

Considering that modern processors can now hash over 3GB per second per core with SHA-256 [1], and considering that blocks are 10 minutes apart on average, there is no major problem to be expected to process blocks of up to 1GB as far as the balanced Merkle tree is concerned. However, as blocks grow larger, the balanced Merkle tree will be an increasing hindrance on scalability.

[1] See https://en.wikipedia.org/wiki/Trie

We propose to replace the balanced Merkle tree by a bitwise trie, applying a Merkle hashing overlay on top of this trie. This strategy is referred to as the *Merklix tree*. The correctness of the Merklix tree depends on a complete ordering of transactions being part of the Bitcoin protocol. The canonical transaction ordering rule [2] grants this property to Bitcoin.

Transaction inclusion proofs remain unchanged form the current balanced Merkle tree, since the Merklix tree uses the same Merkle hashing scheme. It is only the arrangement of nodes that is changed, the internal binary hash construction stays the same. This means that wallets will be unaffected by the change.

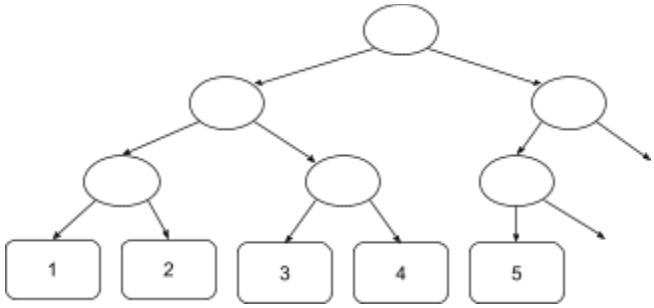
The Merklix tree approach is not novel, and it has been rediscovered multiple times with minor variations by independent inventors who were also trying to improve Bitcoin or a competing alternative to Bitcoin [3] [4] [5] [6].

The Merklix tree lends itself to an incremental embarrassingly parallel implementation for insertions, removals, merges and a variety of other operations. From a Merkle hashing perspective, the Merklix tree provides optimal asymptotic bounds for insertions and removals, with an asymptotic complexity of $O(\ln_2(n))$ if there are *n* transactions in the block. In practice, the constants are also expected to be very low.

From an economic perspective, the Merklix tree preserves all the use cases obtained from the Merkle tree. The Merklix tree lets the *publishers* process larger blocks given a fixed amount of resources invested in hardware. Finally, by eliminating the historical design flaw of the balanced Merkle tree, an implementation tax is removed from the Bitcoin applicative ecosystem.

Balanced Merkle tree

Satoshi's client shapes its Merkle tree as a balanced binary tree, which may contain empty branches or empty leaves on the right side. The construction of the tree is illustrated by the schema below, where 1, 2, ... , 5 refer to the successive transactions to be inserted in the tree.



The Merkle tree is grown from the left to the right. Then, once the tree is complete, an extra level is added at the top, and the process is repeated. The original implementation exhibits a peculiar design flaw as empty branches and empty leaves, always positioned to the right by

construction, are signaled by using the same hash both for the left (non-empty) and right (empty) children.

Using the same hash both right and left to identify empty branch in the Merkle tree is considered a design flaw. This design flaw can be mitigated through dedicated logic. This flaw is well-known and mostly of historical interest, and will not be detailed further[2] as it does not hinder the scaling of Bitcoin. However, fixing this design flaw remains of interest, as it can be seen as an implementation tax imposed on the whole Bitcoin ecosystem for any pieces of software that touch the Merkle tree.

The balanced Merkle tree offers an asymptotic performance of $O(\ln_2(n))$ on insertions when considering a block of n transaction which is asymptotically optimal. However, when considering simple implementations, this performance is only obtained with sequential *append* insertions.

The parallelization of the incremental updates of the balanced Merkle tree is possible, but challenging. For a more detailed discussion of why Bitcoin is concerned with *incremental* updates, that is, the *online* version of the algorithm rather than its *offline* counterpart, we suggest the section *Asymptotic performance analysis* of [2]. In the following, we focus on the online maintenance of the Merkle tree.

If the topological transaction ordering rule is removed from Bitcoin, it becomes possible to shard the workload of the publisher by introducing a series of processing layers. A system with 256 shards, partitioning on the transaction identifiers, could look like the following[3]:

1. *A bounded subtree layer*, which is dedicated to the construction of subtrees of 2^{16} = 65536 transactions. Each shard manages its own set of subtrees, up to 256 subtrees per shard. Each shard ensures that it does not have more than 1 incomplete subtree at any point in time.
2. *A complete subtree reduction layer*, which is dedicated to the consolidation of the incomplete subtrees found in the previous layer into complete subtrees. Considering that there is a maximum of 2^8 = 256 incomplete subtrees to be processed, an 8-pass reduction is sufficient to consolidate all the incomplete subtrees to a set of complete subtrees leaving at most a single incomplete subtree.
3. *A root finalization layer*, which is dedicated to the arrangement of the “top” of the Merkle tree, collecting all the hashes from the complete subtrees as obtained from the two previous layers. Assuming we do not have more than 2^{16} = 65536 subtrees[4], this layer can be operated on a single machine.

[2] See ‘merkle.cpp’ (link) which features a prominent warning about the flaws of the original Merkle tree algorithm, which need to be adequately dealt with.
[3] Those numbers - e.g. number of shards - are provided for the sake of clarity, as the alternative of introducing many variable names is not appealing in this context. Naturally, if such a solution were to be implemented, those numbers would have to be dynamically adjusted depending on the exact blockchain conditions.
[4] With 2^{16} complete subtrees containing 2^{16} transactions, we have about 4 billions transactions in total, which represents a block of 1TB.

If the topological transaction ordering rule is not removed from Bitcoin, then the incremental parallelization of the balanced Merkle tree is even more challenging and would require further processing layers. First, as the transactions that are not built on top of 0-conf can be put anywhere in a block, those transactions would be isolated in order to be processed as detailed above. Second, the transactions built on top of 0-conf would require their own processing layer entangled with the system in charge of delivering an incremental topological ordering. This aspect goes beyond the scope of the present document. Based on the literature presented in [2], such processing layers can be expected to be complex.

Overall, there is little doubt that it *might* be possible to engineer incremental parallel insertions within the balanced Merkle tree with a satisfying degree of parallelism. However, this task is nearly guaranteed to introduce a series of complications in the design of Bitcoin. Furthermore, performance will depend on finely tuned heuristics, which may break or be broken by adversaries. This aspect will require further heuristics intended to mitigate such attack vectors.

This design is not satisfying for infrastructure software which should be made correct-by-design to the greatest extent possible. As we will see in the following, there is a simple solution to remove this problem altogether which has already been reinvented multiple times for Bitcoin or its competitors.

The reinventions of the bitwise trie with Merkle overlay

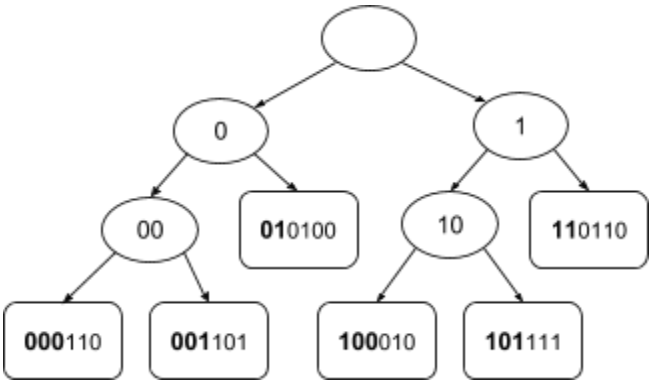
A Merkle tree comes with two major ingredients: a layout for the tree and a hashing function. The hash function used by Bitcoin is a double SHA-256, and discussing alternatives to SHA-256 is beyond the scope of the present document. In terms of tree layouts, the balanced tree complicates any attempt at incremental parallel insertions, precisely because maintaining the balance of the tree requires a coordination mechanism at the tree-level.

Historically, multiple people involved in electronic currencies came to the same conclusion about the flaws of the naive Merkle implementation as originally found in Satoshi's client. Interestingly enough, those people converged to the same solution of leveraging a bitwise trie based on key prefixes to shape their Merkle trees. Those people gave each time a different name for their respective data structures, which hints at multiple independent reinventions. Also, those solutions differ in minor implementation details.

In the Bitcoin community, the concept has been introduced as the *Merkle prefix tree* [3] and the *Merklix tree* [4]. The Ripple community refers to the concept as *hash tree* [5] while the Ethereum community refers to it as *Patricia tree* [6]. Reviewing the minute details of those variants is beyond the scope of the present document.

Fundamentally, the idea consists of organizing the tree as a bitwise trie while using the bits of the keys themselves as prefixes. This approach yields benefits because of the excellent performance of the X-fast trie which have been known for decades [7].

The following schema illustrates how a bitwise trie represents a set of 6 values that happen to be 6 byte long.



The bitwise trie is a binary tree that satisfies the properties:

- a zero (resp. one) node can only be a left (resp. right) child.
- the leaves contain exactly the values of the set with no empty leaves.
- the path to a leaf is a bitwise prefix of the value contained in the leaf.

As the bitwise trie is a regular binary tree, the Merkle hashing is straightforward to apply. Arguably, as the binary tree associated with the bitwise trie is *complete* - i.e. there are no empty leaves - it removes entirely the need to cope with the empty leaf edge case. This property is desirable as its streamlines the implementation of Bitcoin apps.

Insertions with a Merklix tree can be performed in $O(\ln_2(n))$ where n is the number of entries to be inserted, i.e. transactions in the Bitcoin context. Moreover, the Merklix tree trivially lends itself to embarrassingly parallel processing for a wide variety of operations such as insertions or removals, as the data structure can be sharded over arbitrary prefix ranges.

Unlike the balanced Merkle tree, the bitwise trie is an imbalanced tree. However, in the context of Bitcoin, as keys are SHA-256 hashes, the depth of the tree is kept under control due to the very nature of the keys, which are very expensive to craft. Generating an imbalanced tree of depth 256 is equivalent to breaking SHA-256. Thus, the depth of the bitwise trie in the Bitcoin context is guaranteed to have an upper bound at 256. Also, the bound is expected to be much lower in practice.

Replacing the balanced Merkle tree by the Merklix tree

We propose to modify the Bitcoin protocol to change the calculation method for the root hash of the Merkle tree that contains all the transactions. The balanced tree is replaced by the bitwise

trie built from the *transaction identifiers.* The Merkle overlay remains unchanged and computed from the transaction hashes[5]. As the bitwise trie is built from the transaction identifiers, this approach enforces the canonical transaction ordering rule within a block. This bitwise trie with a Merkle overlay is referred to as the Merklix tree.

The pseudo-code for computing the hash of the Merklix tree is given below.

```
// T: set of transactions
// k: index of the bit used for partitioning
MerklixHash(T,k)
{
  match (T) with
    | { t } => SHA256(t) // single transaction
    | (T₀,T₁) where { t ∈ T₀ : t[k] == 0 } and { t ∈ T₁ : t[k] == 1 }
=>
      match (T₀,T₁) with
       | (_,∅) => MerklixHash(T₀,k+1)
       | (∅,_) => MerklixHash(T₁,k+1)
       | _ => SHA256(MerklixHash(T₀,k+1)::MerklixHash(T₁,k+1))
}

// Root start at bit index zero
MerklixHashRoot(T) => MerklixHash(T, 0)
```

The k^{th} bit of the transaction identifier of the transaction `t` is written as `t[k]`. The operator `::` is the byte-wise concatenation. `SHA256` is the double sha-256 used by Bitcoin.

This approach preserves the Merkle tree at the block level, hence it preserves the backward compatibility for all existing pieces of software that navigate a block through the prism of the Merkle tree. The only pieces of software that need to be upgraded are the ones concerning the block hashing logic.

All the compact proofs (e.g. transaction inclusion) that can be established based on the original Merkle tree remain available with the Merklix tree.

[5] At this point in time, transaction identifiers are identical to transaction hashes. The current proposal does not support or oppose MalFix, but we proactively clarify the evolution of the current proposal if MalFix is ever implemented in Bitcoin.

Economics of the Merklix tree

In 10 years since the inception of Bitcoin, no known use case has emerged for the specific layout of the Merkle tree within the blocks. Thus, while reshaping those trees requires a protocol upgrade for *livechain*[6] apps, no existing capability is lost for the ecosystem.

The Merklix tree addresses the challenge of organizing the transactions at scale within a block with minimal computational overhead while preserving all the compact proofs that can be leveraged by *chainless* apps.

Also, by eliminating the design flaw associated with the balanced Merkle tree, as well as eliminating the edge case of the empty branch altogether, the Merklix tree eliminates an implementation tax for the Bitcoin applicative ecosystem.

References

[1] *SHA256 computations in pure Go using AVX512 and AVX2 for Intel and ARM64 for ARM*, Frank Wessels, 2017, (link)
[2] *Canonical Transaction Ordering for Bitcoin*, Joannes Vermorel, Amaury Séchet, Shammah Chancellor, Tomas van der Wansem, June 2018 (pdf)
[3] *Merkle prefix tree*, Mark Friedenbach, 2013, (link)
[4] *Introducing Merklix tree as an unordered Merkle tree on steroid*, Amaury Séchet, September 2016 (link)
[5] *Hash tree*, Ripple documentation, 2013, (link)
[6] *Patricia tree*, Ethereum documentation, 2016, (link)
[7] *Log-logarithmic worst-case range queries are possible in space Θ(N)*, Dan Willard, 1983

[6] See *A taxonomy of the Bitcoin applicative landscape*, Joannes Vermorel, May 2018 (link)