Building more efficient bloom filter for Ethereum blockchain

Naomie Abecassis, Riad Atamny, Sigal Notovich

Link to github repository: https://github.com/sigalnotovich/maxHEAP_GreadyALG_forBloomFilter

1. Introduction

A blockchain is a distributed ledger across the network that is composed of a chain of blocks, each of them containing data. The use of blockchain has been very popularized in the past few years, especially with digital currencies and nowadays there are used in many different fields. One of the most popular blockchain is Ethereum, that uses the Ether as cryptocurrency. The Ethereum platform is the most actively used blockchain [1] and this make every little functionality in it carry a heavy importance. One of the reasons that made Ethereum as popular as it is, is his functionality of smart contract. They allow anyone with programming skills to create and manage actions that happen on the blockchain.

A blockchain, being a decentralized network, needs to handle the coherence of the information over the clients of the blockchain. For that, all the transactions that are made on the blockchain must be known by all the users. This requires sending the data contained in each transaction, and in each block of the blockchain. This important task is called block propagation: whenever a new block is published to the blockchain, it is sent to all the users. This transfer of data is awfully expensive and tends to be a bottleneck in Ethereum blockchain since the most recent transfer technologies still have severe limitations in the time domain, which is such important since it can create conflicts in the blockchain.

In this project we focused on the Ethereum blockchain and proposed a solution to reduce the time required for data transfer during block propagation, by focusing on reducing the number of bits in the bloom filter. In this way we can, first, transfer less data (if transferring only lighten bits) and second, assure a better efficiency of the bloom filter.

2. Bloom filter in Ethereum

As explained above, a blockchain is a chain of blocks where each block contains information stored as transactions. In the Ethereum blockchain, a block typically contains between 150 and 300 transactions, but depends on the gas used by each transaction (the gas is a unit to quantify the computational need of a transaction). In order to make the block propagation more efficient, each block contains a bloom filter with the transactions of the block.

A bloom filter is a space efficient probabilistic data structure that is used to test whether an element is in a set or no. Initially the bloom filter contains only zeros and each action of adding an element to the bloom filter lights up several bits (set the bit to 1) according to a predetermined function. Then, whenever someone wants to check if an element is in the set, he can check if the bits corresponding to the element are set to 1 or no. If one bit (or more) is set to zero it means that the element is not in the set, whereas if all the bits are set to 1 it means that the element is in the set with a probability of α. Indeed, this probabilistic data structure introduces the possibility to have false positives i.e., an element is said to be in the set while it is

not. In the Ethereum blockchain the probability α is computed such that it tends to 1 and thus we can neglect the possible error.

Once a block is published to the blockchain, this block must be transmitted with all his content to the other users. But one interesting point is that whenever a transaction is done, it is also transmitted to the users of the blockchain which keep this transaction in their mempool (a mempool is a set of transactions that were not entered to a block yet). In that way, when a block is published, each user has (almost) all the transactions of the block in his mempool. So, we do not need to send the entire block to each user. With this observation in mind, Ethereum decided to implement an algorithm that will reduce the size of the data that needs to be transferred by sending a bloom filter corresponding to the transactions of the block.

Thereby, when a block is published to the blockchain by a miner, the miner sends the bloom filter of this block to all his neighbors (instead of the block itself). If one of the neighbors discovers that one of the transactions presents in the block is not present in his mempool, he will require from the miner to send the transaction itself. In this way the data to be transmitted is only a bloom filter and in some cases several transactions. This algorithm implemented by Ethereum reduces a lot the amount of data transmitted.

As we can understand from this way of functioning, the efficiency of the bloom filter is a primordial criterion for ensuring the good functioning of the Ethereum blockchain. In this project we propose a solution to achieve high success rate for the bloom filter, meaning that the bloom filter have a lower probability of false positives than original bloom filters.

3. Existing limitations

First, we must explain the existing solution in the Ethereum blockchain for building the bloom filter. To understand this, we had to refer to the Ethereum yellow paper [2]. The size of the bloom filter is set to 2048 bits and each transaction lights up 3 bits “through taking the low-order 11 bits of each of the first three pairs of bytes in a Keccak-256 hash” of the transaction hash. We implemented this function and tested it with real data from the Ethereum blockchain [4] and indeed our observations showed that the error rate is not 0 but some bits in the block are overlapping one with each other. When we started this project, we first checked on real data whether there is way to improve or not. We noticed that the number of bits lighted in the bloom filter to represent a block grows linearly with the number of transactions contained in the block. Indeed, in most cases, each transaction adds 3 new bits to the bloom filter, making the bloom filter less optimal. From that we decided to implement a solution that could reduce the number of bits used to represent the transactions of a bloom filter.

4. Our solution

In this project we propose a solution to improve the success rate of the bloom filter. We want to guarantee that each transaction that we add to the bloom filter will set to 1 a minimal number of bits. In that way the bloom filter will be constructed such that it will reduce false positives. To answer this challenge, we propose two solutions: the first one based on a greedy algorithm, and the second one implemented with a dynamic programming algorithm. Moreover, we found efficient to add an optimization to the greedy algorithm, as explained in part b below.

a. Greedy algorithm

The first solution that we propose is to use a greedy algorithm. In each step, the algorithm chooses the bit that is the most lighted up by all the transactions in the mempool.

Algorithm:

1. We construct set of transactions to be added the block.
2. For each transaction to be added to the block, do:
   2.1. If the transaction raises bits that are not already '1' in the bloom filter we discard the transaction.

   2.2. Else, add it to the block.

Input: A mempool of Ethereum transactions.

Output: An order of the transactions to be added to the blocks.

Note: The verification is made by bitwise-xor on the bits that the transaction raises, denote the result ABC then we perform an AND operation RESULT=AND(A,B,C) and if RESULT='1' we add it to the block, else the transaction is discarded.

b. Optimized greedy algorithm

The approach of this algorithm is determining the number of lighted bits in each transaction by creating a histogram and a max heap that returns the bits that are used each time with a pointer to the transactions which are related to these bits.

Algorithm:

1. We construct a histogram for the 2048 bits, with pointers to the transactions like Union-Find.
2. Until the block is full, do:
   2.1. Pop the bit with maximum value in the histogram.
      1. Return all the transactions associated with the bit and add them to the block.

Input: A mempool of Ethereum transactions

Output: An order of the transactions to be added to the blocks

c. Dynamic programming

We take the same algorithm as in b. and instead of directly adding the transactions to the block we perform a dynamic step. Backtracking the effect on the number of '1' bits overall, after taking or not taking the transaction.

The general formula implemented in this algorithm is:

$$T(i,n) = \begin{cases} \min\{T(i,n-1) + a_i, T(i+1,n)\}, & i < m \text{ and } n > 0 \\ \infty, & i = m \text{ or } n > 0 \\ 0, & n = 0 \end{cases}$$

Where $T(i,n)$ corresponds to the minimum number of bits lighted by taking i transactions among a mempool containing n transactions, a_i is the number of new bits lighted by the transaction i (in the current bloom filter), and m is the required number of transactions (that we want to enter to the bloom filter).

Note: we chose not to implement the algorithm since such implementation took way too much time to run even on the faculty servers due to recursive backtracking with the immense amount of data (still the code can be found in the git repository). Therefore, we suggest an improved implementation as further work.

5. Results

In order to test our implementations, we downloaded few blocks from etherscan.io website. These blocks are actual blocks from the Ethereum blockchain, so we can prove that our solution works for real data values.

The results show an improvement of up to 24% in the quantity of bits that are light up in each bloom filter of the blocks. In the figure 1 below, we show the number of lighted bits before and after running the algorithm according to the size of the mempool. As we can see, we achieve a maximal improvement for a mempool of 1050 transactions, decreasing by 24% the number of '1's in the bloom filter.

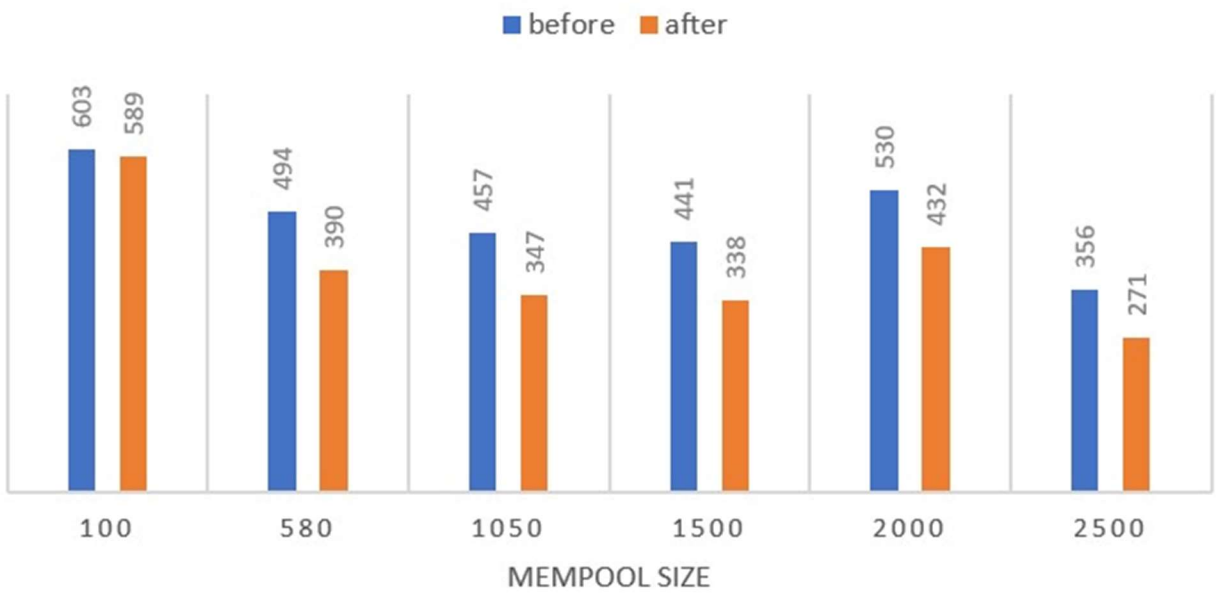

Figure 1. Chart showing the number of lighted bits before/after running the algorithm according to the mempool size

We ran this experiment on a large amount of blocks too, and the results are shown in figure 2 below.

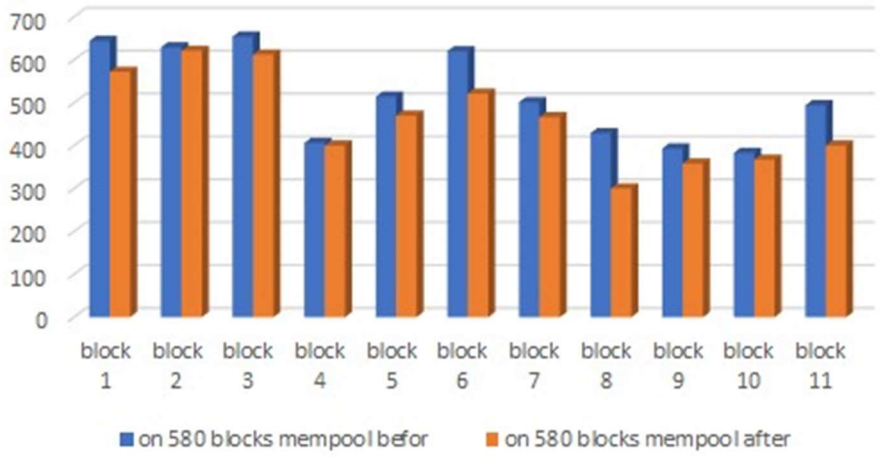

Figure 2. Chart showing the number of lighted bits before/after running the algorithm on an experiment of 580 blocks

These two graphs clearly show an important improvement in the size of the bloom filter that represents a block, showing us the importance of how we choose the transactions in a certain block. As an indication, the number of transactions on the Ethereum blockchain every day is around 1.3M transactions, and the number of blocks that are deployed every day to the Ethereum blockchain is around 6,000 blocks. These numbers show the importance of such a work, having a great impact on the data transferred on the blockchain.

6. Conclusion and future work

As a conclusion, we can say that our implementation improves the efficiency of the bloom filter used in the Ethereum blockchain and allows us to transfer less data in the blockchain. This has a real importance knowing the speed of extension of the Ethereum blockchain nowadays.

We could go further in that research work by improving the complexity of the algorithm used, or by adding a constraint to maximize the profit that the miner gets when publishing a block. In that way we could choose the best combination of transaction to be the one that contains ligths up minimum number of bits and has the minimum fee value, which will give to the miner a better option for his benefits.

7. References

1. *https://en.wikipedia.org/wiki/Ethereum*
2. *Wood, G. (2014). Ethereum: A secure decentralized generalized transaction ledger. Ethereum project yellow paper, 151(2014), 1-32.*
3. *https://en.wikipedia.org/wiki/Bloom_filter*
4. https://etherscan.io/

8. Appendix

At the beginning our first implementation was a min max flow algorithm, to maximize the flows. Unfortunately, we understood (after implementation...) that it was not a satisfactory solution, so, we add to change it.

Min-Cut Max-Flow algorithm

This solution relies on a reduction to the maximum flow algorithm. Considering a block with n transactions, denoted as $t1, t2, \dots, tm$, we build a flow grid in that way:

- The group of nodes contains (layer by layer):
  i. First Layer denoted S is 2048 nodes corresponding to the number of bits in the bloom filter denoted $si, 1 < i < 2048$.
  ii. Second Layer denoted S' is again 2048 nodes corresponding to the number of the nodes in the first layer denoted $s'i, 1 < i < 2048$, as bits raised by txns in the third layers.
  iii. Third layer denoted V will be the m nodes corresponding to the number of transactions in the txn pool denoted $vi, 1 < i < m$.
  iv. Fourth layer $\{t'\}$ will be a single target node, prior to the actual target denoted t'.
  v. Fifth and last layer {t} will be the actual target node denoted t.

- The group of edges contains:
  i. $SS' = \{(s_i, s'_i) \mid 1 < i < 2048, \ and \ c((s_i, s'_i)) = 1\}$
  ii. $S'V = \{(s'_i, v_j) \mid 1 < i < 2048, \ and \ \boldsymbol{i \ is \ the \ bit \ raised \ to \ 1 \ by \ transaction \ j}, \ 1 < j < m \ and \ c((s'_i, v_j)) = 1\}$
  iii. $VT' = \{(vi, t') \ for \ each \ vi \ in \ V \ c \ (vi, t') = 3.$
  iv. $T'T = \{(t', t) \ where \ c \ (t', t) = c$ that is determined by c=3*n as n is the number of transactions to be taken according to the gas price limitation}.

Input: A graph $G(V, E)$ with edge capacity function.

Output: Transactions to pick for the bloom filter.

Algorithm:

a. Build flow net N(G,s,t,c).
b. Run Ford-Fulkerson algorithm.
c. Pick min-cut {A}, V\{A} where A = {t',t}.
d. For each edge in cut {A}, V\{A} -> e = (v_i,t') where f(e)=c(e)=3 and v_i is in V add txn_i to block.

We did not find a way to adapt this solution to our research, so we put it aside.