Transaction Scheduling for Solana

July 15, 2022

1 Model

Solana faces the following scheduling problem in parallelizing the execution of transactions:

- We start by considering the scheduling problem for a single block with a fixed pool of transactions T [1]
- Each transaction i consists of
  - f_i, A transaction fee;
  - c_i, How many CU[2] the txn consumes[3];
  - R_i, a list of accounts it will read from;
  - W_i, a list of accounts it will write to;
- There are k threads to schedule jobs on with each thread having a capacity C of maximum CU units the thread can schedule per block
- If transaction i writes to account a, then no other transaction j that either reads from a or writes to a can be processed at the same time i is being processed [4]
- We seek to maximize the total revenue that can be included in a block given these constraints [5]

While we start with this simplified model, we note the following technical difficulties not covered

- The scheduler sending a single transaction at a time to each thread to be processed is inefficient. It is better engineering wise for batches of transactions to be sent to threads for completion. The implication being, given c_i as a noisy signal, it may not be optimal to wait for single transactions to finish processing to have exact knowledge of completion time before sending that thread more transactions.

[1] In reality this pool is dynamically updating with transactions arriving on the fly and transactions expiring after waiting long enough to be scheduled but we start by analyzing the static case

[2] CU is the equivalent of gas in Solana

[3] This cost is a correlated signal for time the transaction takes to process but does not provide exact estimates

[4] Note if both i and j are only reading from a and not writing to a then it is fine for them to be simultaneously scheduled across different threads

[5] Solana can't force any validator to use a scheduler so it's ideal that this scheduler is incentive compatible for validators to run

- In order to maximize throughput, it is desirable that threads have a constant queue of transactions they can process instead of ever needing to wait for the scheduler to send them new transactions upon completion of old transactions
- In our model, the scheduler has a global view over the state of each of the threads at any given time, but in reality the scheduler only gains information when threads send the scheduler information
  - There is some overhead to sending this information so minimizing the amount of info needing to be sent is ideal
  - The current implementation idea in Solana is that a thread completes processing a batch of transactions and sends the completed batch back to the scheduler as a signal it has finished processing those transactions.

2 Current Solana Design

The current design idea Solana's team has for their scheduler is as follows

1. While a transaction t hasn't been processed, it is in one of three states *scheduled, pending,* or *blocked*. Transactions are correspondingly stored in the sets S,P, or B depending on which state they are in. P maintains the invariant that all of its transactions are always listed in decreasing order according to the ratio f_i/c_i.
2. Initially S and B are empty with all the transactions starting in P.
3. The scheduler then builds a batch Q_i in the following greedy order
   (a) The scheduler adds the first transaction t in P to Q_i. Now t changes its state to scheduled and is moved from P to S. Then any transactions t would conflict with in P become blocked and are moved to B. Formally, for every other transaction $t' \in P$, if $W_t \cap (W_{t'} \cup R_{t'}) \neq \emptyset$ or $R_t \cap W_{t'} \neq \emptyset$ then t' is moved from P to B and becomes blocked.
   (b) The scheduler proceeds greedily adding transactions from the top of P to Q_i and updating conflicting transactions' state to be blocked as necessary until the batch passes some threshold G of total CU
4. The scheduler then repeatedly sends a batch Q_i to a thread and builds a new batch Q_{i+1} until each thread has at least 2 batches scheduled for processing [6][7].
5. When a thread k finishes processing a batch Q_i of transactions, k removes Q_i from its queue and sends the scheduler a completion signal
6. Upon receiving a completion signal for Q_i from a thread k, the scheduler updates $S \leftarrow S \setminus Q_i$. The scheduler then checks which transactions in B are no longer being blocked by any transactions in S and moves such transactions back to P[8].

[6] Once a thread receives a batch, it starts processing immediately and continues processing until it runs out of transactions.

[7] If every thread has at least 2 batches in its queue the scheduler pauses and waits for a completion signal from one of the threads

[8] This is done efficiently by having a Hashmap for all the transactions a given transaction is blocking and keeping track of the minimum priority transaction scheduled that touches any given account, details in https://github.com/solana-labs/solana/pull/26362

7. The scheduler then repeats Step 3 to build a new batch Q_j. If processing Q_j would not push thread k past processing more than C CU, then the scheduler sends Q_j to k's batch queue.

8. Steps 5-7 are repeated until a block is produced.

In the model where T is dynamically updating , when a transaction t arrives, if there is a transaction $t' \in S$ that conflicts with t then t is added to B otherwise t is added to P. Additionally note that this implementation implies that conflicting transactions can't be scheduled on the same thread until one of the conflicting transactions is fully processed. This is unnecessary since transactions that read/write to the same accounts can be safely executed if they are executed on the same thread serially. This is looking to be improved by keeping track of which threads a blocked transaction has conflicts with and allowing conflicting transactions to be scheduled on the same thread.

Without this change the this schedule can have have arbitrarily bad approximation with respect to the optimal revenue if all the highest value transactions are conflicting with each other allowing only one of these transactions per batch on one thread. With this change, my intuition is that the worst case performance of this greedy approach is $\frac{1}{k}$ when all the c_i are small with respect to C. The worst case happens when there is one set of transactions that conflict with every other transaction but only have marginally higher ratios of f_i/c_i. Then all these transactions get greedily scheduled on the same thread and block any other transactions from being scheduled on any of the other threads. In general this suggest the problem with the greedy approach is when transactions that have marginally higher fees but many conflicts are scheduled with higher priority than similar fee transactions with very few conflicts.