Polkadot Weights

Web3 Foundation

April 2020

1 Motivation

Polkadot has a limited time window for block producers to create a block, including limitations on block size which can make the selection and execution of certain extrinsics too expensive and decelerate the network. The weight system introduces a mechanism for block producers to measure the expense of extrinsics and determine how "heavy" it is. With this mechanism, block producers can select a set of extrinsics and saturate the block to it's fullest potential without exceeding any limitations (as described in section 3).

Polkadot also introduces a specified block ratio (as defined in section 3), ensuring that only a certain portion of the total block size gets used for regular extrinsics. The remaining space is reserved for critical, operational extrinsics required for the functionality by Polkadot itself.

2 Fundamentals

Weights are just a numeric value and Runtime functions may use complex structures to express those values. Therefore, the following requirements must apply for implementing weight calculations:

- Computations of weights must be determined before execution of that extrinsic.
- Due to the limited time window, computations of weights must be done quickly and consume few resources themselves.
- Weights must be self contained and must not require I/O on the chain state. Weights are fixed measurements and are based solely on the Runtime function and its parameters.
- Weights serve three functions: measurements used to calculate transaction fees, to prevent the block being filled with too many extrinsics and to avoid extrinsics where its execution takes too long.

3 Limitations

The assigned weights should be relative to each others execution time and "heaviness", although weights can be assigned depending on the priorities the chain is supposed to endorse. Following limitations must be considered when assigning weights, which vary on the Runtime.

3.1 Considerable limitations

- Maximum block length
- Maximum block weight
- Targeted time per block
- Available block ration reserved for normal, none-operational transactions

3.2 Considerable limitations in Polkadot

As of the official Polkadot Runtime, the limitations are set as follows:

- Maximum block length: $5 \times 1'024 \times 1'024 = 5'242'880$ Bytes
- Maximum block weight: 1'000'000'000
- Targeted time per block: 6 seconds
- Available block ratio: 75%

The values of the assigned weight itself is not relevant. It must only fulfill the requirements as noted by the fundamentals and limitations, and can be assigned as the author sees fit. As a simple example, consider a maximum block weight of 1'000'000'000, an available ratio of 75% and a targeted transaction throughput of 500 transactions, we could assign the weight for each transaction at about 1'500'000.

Do note that the smallest, non-zero weight in Polkadot is set at 10'000.

4 Weight Assignment

Assigning weights based on theoretical performance such as big O notation proves to be unreliable and too complex due to imprecision in back-end systems, internal communication within the Runtime and design choices in the software. Therefore, all available Runtime functions, which create and execute extrinsics, have to be benchmarked with a large collection of input parameters.

4.1 Parameters

The inputs parameters highly vary depending on the Runtime function and must therefore be carefully selected. The benchmarks should use input parameters which will most likely be used in regular cases, as intended by the authors, but must also consider worst case scenarios and inputs which might decelerate or heavily impact performance of the function. The input parameters should be randomized in order to cause various effects in behaviors on certain values, such as memory relocations and other outcomes that can impact performance.

It's not possible to benchmark every single value. However, one should select a range of inputs to benchmark, spanning from very small values (or none, if the Runtime function allows it) to large values which will most likely exceed the expected usage of that function. As an example, considering a Runtime function which transfers balances to one or multiple accounts, one could select a range spanning from very small balances being sent from one to hundreds of accounts, to very large balances being sent from one to hundreds of accounts.

The benchmarks should run individual executions/iterations within that range, where the chosen parameters should give insight on an average execution time and resource cost. Selecting imprecise parameters or too extreme ranges might indicate an inaccurate average of the function as it will be used in production. Therefore, when a range of input parameters gets benchmarked, the result of each individual parameter should be recorded and ideally visualized. The author should then decide on the most probable average execution time, basing that decision on the limitations of the Runtime and expected usage of the network.

Additionally, given the distinction between theory and practice, the author reserves the right to make adjustments to the input parameters and assigned weights according to observed behavior of the actual, real-world network.

4.2 Blockchain State

The benchmarks should be performed on blockchain states that already polluted and contain a history of extrinsics and storage changes. Runtime functions that require read/writing on structures such as Tries will therefore produce more realistic results that will reflect the real-world performance of the Runtime.

4.3 Environment

The benchmarks should be executed on clean systems without interference of other processes or software. Additionally, the benchmarks should be executed multiple machines with different system resources, such as CPU performance, CPU cores, RAM and storage speed.

4.4 Preliminary Work

In order to produce conditions where resource heavy computation or excessive IO can be observed, the benchmarks might require some preliminary work on the environment. As a practical example, this section describes the specifically designed benchmark of the `transfer` function available in the Polkadot Runtime. The benchmark is designed to measure the functions worst possible condition:

- Transfer will kill the sender account (by completely depleting the balance to zero).
- Transfer will create the recipient account (the recipient account doesn't have a balance yet).

Parameters

The following parameters are selected:

Type		**From**	**To**	**Description**
Account index	`index` in...	1	1000	Used as a seed for account creation
Balance	`balance` in...	2	1000	Sender balance and transfer amount

Executing a benchmark for each balance increment within the balance range for each index increment within the index range will generate too many variants (1000×999) and highly increase execution time. Therefore, this benchmark is configured to first set the balance at value 1'000 and then to iterate from 1 to 1'000 for the index value. Once the index value reaches 1'000, the balance value will reset to 2 and iterate to 1'000 (see algorithm 1 for more detail):

- `index`: 1, `balance`: 1000
- `index`: 2, `balance`: 1000
- `index`: 3, `balance`: 1000
- ...
- `index`: 1000, `balance`: 1000
- `index`: 1000, `balance`: 2
- `index`: 1000, `balance`: 3
- `index`: 1000, `balance`: 4
- ...

The parameters itself do not influence or trigger the two worst conditions and must be handled by the implemented benchmarking tool. The $transfer$ benchmark is implemented as defined in algorithm 1.

Implementation

The benchmarking implementation for the Polkadot Runtime function $transfer$ is defined as follows:

Algorithm 1: Run multiple benchmark iterations for $transfer$ Runtime function

```
Result: A collection of time measurements of all benchmark iterations
Init: collection = {};
Function Main is
    balance = 1'000;
    for index ← 1 to 1'000 increment by 1 do
        time ← Run-Benchmark(index, balance);
        Add-To(collection, time);
    end
    index = 1'000;
    for balance ← 2 to 1'000 increment by 1 do
        time ← Run-Benchmark(index, balance);
        Add-To(collection, time);
    end
end
Function Run-Benchmark(index, balance) is
    sender ← Create-User("caller", index);
    recipient ← Create-User("recipient", index);
    Set-Balance(sender, balance);
    time ←Timer(Transfer-Balance(sender, recipient, balance));
    return time
end
```

- Add-To(*collection*, *time*)

  - Adds a returned time measurement (*time*) to *collection*.

- Create-User(*name*, *index*)

  - Creates a Blake2 hash of the concatenated input of *name* and *index* representing the address of a account.

- Set-Balance(*address*, *balance*)

  - Sets a initial *balance* for the specified account (*address*) in the storage state.

- Timer($function$)

  - Measures the time from the start of the specified $function$ to its completion.

- Transfer-Balance(*sender*, *recipient*, *balance*)

  - Transfers the specified *balance* from *sender* to *recipient* by calling the corresponding Runtime function.

5 Fees

Block producers charge a fee in order to be economically sustainable. That fee must always be covered by the sender of the transaction. Polkadot has a flexible mechanism to determine the minimum cost to include transactions in a block.

5.1 Fee Calculation

Polkadot fees consists of three parts:

- Base fee: a fixed fee that is applied to every transaction and set by the Runtime.
- Length fee: a fee that gets multiplied by the length of the transaction, in bytes.
- Weight fee: a fee for each, varying Runtime function. Runtime implementers need to implement a conversion mechanism which determines the corresponding currency amount for the calculated weight.

The final fee can be summarized as:

$$\begin{aligned} fee = & \; base\ fee \\ & + length\ of\ transaction\ in\ bytes \times length\ fee \\ & + weight\ to\ fee \end{aligned}$$

5.2 Definitions in Polkadot

The Polkadot Runtime defines the following values:

- Base fee: 100 uDOTs
- Length fee: 0.1 uDOTs
- Weight to fee conversion:

  $$weight\ fee = weight \times (100\ uDOTs \div (10 \times 10'000))$$

  A weight of 10'000 (the smallest non-zero weight) is mapped to $\frac{1}{10}$ of 100 uDOT.
  This fee will never exceed the max size of an unsigned 128 bit integer.

5.3 Fee Multiplier

Polkadot can add a additional fee to transactions if the network becomes too busy and starts to decelerate the system. This fees can create incentive to avoid the production of low priority or insignificant transactions. In contrast, those additional fees will decrease if the network calms down and it can execute transactions without much difficulties.

That additional fee is known as the `Fee Multiplier` and its value is defined by the Polkadot Runtime. The multiplier works by comparing the saturation of blocks; if the previous block is less saturated than the current block (implying an uptrend), the fee is slightly increased. Similarly, if the previous block is more saturated than the current block (implying a downtrend), the fee is slightly decreased.

The final fee is calculated as:

$$final\ fee = fee \times Fee\ Multiplier$$

5.3.1 Update Multiplier

The `Update Multiplier` defines how the multiplier can change. The Polkadot Runtime internally updates the multiplier after each block according the following formula:

$$
\begin{aligned}
diff &= (target\ weight - previous\ block\ weight)\\
v &= 0.00004\\
next\ weight &= weight \times (1 + (v \times diff) + (v \times diff)^2/2)
\end{aligned}
$$

Polkadot defines the `target_weight` as 0.25 (25%). More information about this algorithm is described in the Web3 Foundation research paper: `https://research.web3.foundation/en/latest/polkadot/Token%20Economics.html#relay-chain-transaction-fees-and-per-block-transaction-limits`.