

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 its 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: $2'000'000'000'000$
- 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 Runtime Primitives

The Runtime functions must be studied in order to determine which parts of the code will excessively increase execution time. Potential indicators like loops, IO operations and data manipulation must be considered. Based on those observations, Weights should be applied on operations that could have heavy implications on those design choices.

Section 6.1 walks through two practical examples of such analysis. Not every possible outcome can be caused with input parameters to the Runtime function. In some circumstances, preliminary work regarding storage is required before a specific benchmark can be reliably measured. Two of such examples are described in section 6.2.

4.1 Primitive Types

The Runtime reuses components/primitives to interact with the state storage. The behavior and the execution cost of those primitives can be studied and a Weight should be applied for each use within the Runtime code.

For storage, Polkadot uses three different types of storage types across its modules, depending on the context:

- **Value:** Operations on a single value.

The final key-value pair is stored under the key:
`hash(module_prefix) + hash(storage_prefix).`

- **Map:** Operations on multiple values, datasets, with their corresponding, unique keys.

The final key-value pair is stored under the key:
`hash(module_prefix) + hash(storage_prefix) + hash(encode(key)).`

- **Double map:** Just like **Map**, but uses two keys instead of one (child storages).

The final key-value pair is stored under the key:
`hash(module_prefix) + hash(storage_prefix) + hash(encode(key1)) + hash(encode(key2)).`

Which type to use depends on the functionality of the Runtime module (or its sub-processes, rather). In some cases only a single value is required. In others, multiple values need to be operated on. The types **Map** and **Double map** support multiple hashing algorithms, each providing different properties in terms of safety (collision resistance) and performance:

- Blake2 128-bit concat (hash + key concatenated)
- xxHash 64-bit concat (hash + key concatenated)
- Un-hashed, raw key

Those lower level types get abstracted over in each individual Runtime module using the `decl_storage!()` macro. Therefore, each module specifies its own types that are used as input and output values. The abstractions do give indicators on what operations must be closely observed and where potential performance penalties and attack vectors are possible.

4.1.1 Considerations

The storage layout is mostly the same for every primitive type, primarily differentiated by using special prefixes for the storage key. Big differences arise on how the primitive types are used in the Runtime function, on whether single values are operated on or entire lists of datasets. Single value operations are generally quite cheap and its execution time does not vary depending on the data that's being processed. However, excessive IO overhead can appear when IO operations are executed repeatedly, such as loops. Especially, when the amount of loop iterations can be influenced by the caller of the function or by certain conditions in the state storage.

Maps, in contrast, can have additional overhead when inserting or retrieving lists of datasets. Additionally, the Runtime function has to process each item inside that list.

Indicators for performance penalties:

- **Fixed iterations and datasets** - Fixed iterations and datasets can increase the overall cost of the Runtime functions, but the execution time does not vary depending on the input parameters or storage entries. A base Weight is appropriate in this case.
- **Adjustable iterations and datasets** - If the amount of iterations or datasets depend on the input parameters of the caller or specific entries in storage, then a certain Weight should be applied for each (additional) iteration or item. The Runtime defines the maximum value for such cases. If it doesn't, it unconditionally has to and the Runtime module must be adjusted.

When selecting parameters for benchmarking, the benchmarks should range from the minimum value to the maximum value.

Note: What the maximum number should be really depends on the functionality that the Runtime function is trying to provide. If the choice for that number is not obvious, then it's advised to run benchmarks on a big range of values and pick a conservative number below the **targeted time per block** limit as described in section 3.

- **Input parameters** - Input parameters that users pass on to the Runtime function can result in dangerous consequences. Depending on the data type, it can be appropriate to add additional Weights based on certain properties, such as data size. The Runtime must define limits of those properties. If it doesn't, it unconditionally has to and the Runtime module must be adjusted.

When selecting parameters for benchmarking, the benchmarks should range from small values to the maximum value.

5 Benchmarking

Assigning weights based on theoretical performance can 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.

In order to select useful parameters, the Runtime functions have to be analysed in order to understand which behaviors or conditions can result in expensive execution times, as described in section 4.1.

5.1 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 the minimum value to the maximum value which will most likely exceed the expected usage of that function. This is described in more detail in section 4.1.

The benchmarks should run individual executions/iterations within that range, where the chosen parameters should give insight on the execution time and resource cost. Selecting imprecise parameters or too extreme ranges might indicate an inaccurate result 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 theoretical and practical usage, 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.

5.2 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 I/O on structures such as Tries will therefore produce more realistic results that will reflect the real-world performance of the Runtime.

5.3 Environment

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

6 Practical examples

6.1 Runtime function analysis

This section walks through Runtime functions available in the Polkadot Runtime to demonstrate the analysis process as described in section 4.1.

6.1.1 Practical example #1

Analysis

In Polkadot, accounts can save information about themselves onchain, known as the "Identity Info". This includes information such as display name, legal name, email address and so on. Polkadot selects a set of registrars which can judge identities and therefore incentivizes a reputation model. The judgement itself is done offchain. The registrars rating, however, is saved onchain, directly in the corresponding Identity Info. It's also note worthy that Identity Info can contain additional fields, set manually by the corresponding account holder.

The function `request_judgement` from the `identity` Pallet allows users to request a judgement from a specific registrar. Studying this function reveals multiple design choices that can impact performance.

First, it fetches a list of current registrars from storage and then searches that list for the specified registrar index.

```
let registrars = <Registrars<T>>::get();
let registrar = registrars.get(reg_index as usize).and_then(Option::as_ref)
    .ok_or(Error::<T>::EmptyIndex)?;
```

Then, it searches for the Identity Info from storage, based on the sender of the transaction.

```
let mut id = <IdentityOf<T>>::get(&sender).ok_or(Error::<T>::NoIdentity)?;
```

The Identity Info contains the entirety of entries, in an ordered form. It then proceeds the search all those entries for the specified registrar index. If an entry can be found, the value is updated (assuming the registrar is not "stickied", which implies it cannot be changed). In the context of registrars, this update implies that the Identity Info should be rejudged. If the entry cannot be found, the value is inserted into the index where a matching element could be inserted while maintaining sorted order. This results in memory reallocation.

```
match id.judgements.binary_search_by_key(&reg_index, |x| x.0) {
    Ok(i) => if id.judgements[i].1.is_sticky() {
        Err(Error::<T>::StickyJudgement)?
    } else {
        id.judgements[i] = item
    },
    Err(i) => id.judgements.insert(i, item),
}
```

After that, it proceeds to reserve the registrar fee, inserts the newly updated identity info into storage and deposits the event into the scheduler.

```
T::Currency::reserve(&sender, registrar.fee)?;
<IdentityOf<T>>::insert(&sender, id);
Self::deposit_event(RawEvent::JudgementRequested(sender, reg_index));
```

Considerations

Based on this analysis, the following notable conclusions can be drawn:

- The list of registrars varies. Fetching and searching through it can impact execution time. Given that only one operation is required to fetch the account, no matter the input, a too excessive deviation is not to be expected.

- The benchmarks should be executed with different list sizes, from only a few to many registrars.
- The specified sender of the transaction is fetched from storage. Depending on the total accounts available in the storage, the execution time can vary. Given that only one operation is required to fetch the account, no matter the input, a too excessive deviation is not to be expected.
 - The benchmarks should be executed with a different amount of preexisting accounts in storage.
- The specified registrar is searched for in the Identity Info. Additionally, if a new value gets inserted into the byte array, memory get reallocated. Depending on the size of the Identity Info, the execution time can vary.
 - The benchmarks should be executed with different sizes of the Identity Info, including additional fields.
 - The benchmarks should request judgement from registrars that are already inserted in the Identity Info including newly introduced ones, in order to trigger memory reallocation.
 - It is legitimate to introduce additional Weights for changes the sender has influence over, such the additional fields in the Identity Info.

6.1.2 Practical example #2

Analysis

The function `payout_stakers` from the `staking` Pallet can be called by a single account in order to payout the reward for all nominators who back a particular validator. The reward also covers the validator's share. This function is interesting because it iterates over a range of nominators, which varies, and does IO operation for each of them.

First, this function makes some basic checks to verify if the specified era is not higher than the current era (future) and is within the allowed range ("history depth"), specified by the Runtime. After that, it fetches the era payout from storage and additionally verifies whether the specified account is indeed a validator and receives the corresponding "Ledger".

```
let era_payout = <ErasValidatorReward<T>>::get(&era)
    .ok_or_else(|| Error::<T>::InvalidEraToReward)?;

let controller = Self::bonded(&validator_stash).ok_or(Error::<T>::NotStash)?;
let mut ledger = <Ledger<T>>::get(&controller).ok_or_else(|| Error::<T>::NotController)?;
```

The Ledger keeps a list of tracked rewards. The function only retains the entries of the "history depth", and conducts a binary search for the specified era.

```
ledger.claimed_rewards.retain(|&x| x >= current_era.saturating_sub(history_depth));
match ledger.claimed_rewards.binary_search(&era) {
    Ok(_) => Err(Error::<T>::AlreadyClaimed)?,
    Err(pos) => ledger.claimed_rewards.insert(pos, era),
}
```

The retained claimed rewards are inserted back into storage.

```
<Ledger<T>>::insert(&controller, &ledger);
```

The Runtime is actually optimized to some degree: it only fetches a list of the highest staked nominators, a maximum of 64. The rest gets no reward.

```
let exposure = <ErasStakersClipped<T>>::get(&era, &ledger.stash);
```

Next, the function gets the era reward points from storage.

```
let era_reward_points = <ErasRewardPoints<T>>::get(&era);
```

After that, the payout is split among the validator and its nominators. The validators receives the payment first, creating an insertion into storage and sending a deposit event to the scheduler.

```
if let Some(imbalance) = Self::make_payout(
    &ledger.stash,
    validator_staking_payout + validator_commission_payout
) {
    Self::deposit_event(RawEvent::Reward(ledger.stash, imbalance.peek()));
}
```

Then, the nominators receive a payout. The functions loops through the nominator list, conducting a insertion into storage and a creation of a deposit event for each of the nominators.

```
for nominator in exposure.others.iter() {
    let nominator_exposure_part = Perbill::from_rational_approximation(
        nominator.value,
        exposure.total,
    );

    let nominator_reward: BalanceOf<T> = nominator_exposure_part * validator_leftover_payout;
    // We can now make nominator payout:
    if let Some(imbalance) = Self::make_payout(&nominator.who, nominator_reward) {
        Self::deposit_event(RawEvent::Reward(nominator.who.clone(), imbalance.peek()));
    }
}
```

Considerations

Based on this analysis, the following notable conclusions can be drawn:

- The Ledger contains a list of claimed rewards. Fetching, retaining and searching through it can affect execution time. The retained list is inserted back into storage.
 - The benchmarks should be executed with different list sizes, from only a few to many claimed rewards.
- The function searches the database for the reward points of the specified validator. If there are a lot of validators in the network, the search becomes more expensive.
 - The benchmarks should be executed with different amounts of active validators.
- Looping through a list of nominators and creating IO operations for each heavily increases execution time. The Runtime fetches up to 64 nominators.
 - The benchmarks should be executed with different amounts of nominators that back an individual validator. The more there are, the higher is the IO overhead.

6.2 Preliminary Work

In order for certain benchmarks to produce conditions where resource heavy computation or excessive I/O can be observed, the benchmarks might require some preliminary work on the environment, since those conditions cannot be created with simply selected parameters. As practical examples, this section describes the specifically designed benchmarks for the **transfer** and **withdraw_unbonded** functions available in the Polkadot Runtime.

6.2.1 Practical example #1

The *transfer* function of the *balances* module is designed to move the specified balance by the sender to the receiver. The benchmark is configured to measure the function's 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 2 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 2.

Implementation

The benchmarking implementation for the Polkadot Runtime function *transfer* is defined as follows (starting with the MAIN function):

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

Result: *collection*: a collection of time measurements of all benchmark iterations

Function MAIN is

```
Init: collection = {};  
Init: balance = 1'000;  
for index  $\leftarrow$  1 to 1'000 increment by 1 do  
  | time  $\leftarrow$  RUN-BENCHMARK(index, balance);  
  | ADD-TO(collection, time);  
end  
Init: index = 1'000;  
for balance  $\leftarrow$  2 to 1'000 increment by 1 do  
  | time  $\leftarrow$  RUN-BENCHMARK(index, balance);  
  | ADD-TO(collection, time);  
end
```

end

Function RUN-BENCHMARK(*index*, *balance*) is

```
sender  $\leftarrow$  CREATE-ACCOUNT("caller", index);  
recipient  $\leftarrow$  CREATE-ACCOUNT("recipient", index);  
SET-BALANCE(sender, balance);  
time  $\leftarrow$  TIMER(TRANSFER(sender, recipient, balance));  
return time
```

end

- CREATE-ACCOUNT(*name*, *index*)
 - Creates a Blake2 hash of the concatenated input of *name* and *index* representing the address of a account. This function only creates an address and does not conduct any I/O.
- SET-BALANCE(*account*, *balance*)
 - Sets a initial *balance* for the specified *account* in the storage state.
- TRANSFER(*sender*, *recipient*, *balance*)
 - Transfers the specified *balance* from *sender* to *recipient* by calling the corresponding Runtime function. This represents the target Runtime function to be benchmarked.
- ADD-TO(*collection*, *time*)
 - Adds a returned time measurement (*time*) to *collection*.
- TIMER(*function*)
 - Measures the time from the start of the specified *function* to its completion.

6.2.2 Practical example #2

The *withdraw_unbonded* function of the *staking* module is designed to move any unlocked funds from the staking management system to be ready for transfer. The benchmark requires a couple of I/O operations:

- Create stash account and set initial balance.
- Create controller account and set initial balance.

- Bond a certain amount of the funds.
- Unbond full amount of the funds.
- Withdraw unbonded amount, making it ready for transfer.

Parameters

The following parameters are selected:

Type	From	To	Description
Account index	index in...	0 1000	Used as a seed for account creation

This benchmark does not require complex parameters. The values is use solely for account generation.

Implementation

The benchmarking implementation for the Polkadot Runtime function *withdraw_unbonded* is defined as follows:

Algorithm 2: Run multiple benchmark iterations for *transfer* Runtime function

Result: *collection*: a collection of time measurements of all benchmark iterations

Function MAIN is

```

Init: collection = {};
for index  $\leftarrow$  0 to 1'000 increment by 1 do
  stash  $\leftarrow$  CREATE-ACCOUNT("stash", index);
  controller  $\leftarrow$  CREATE-ACCOUNT("controller", index);
  SET-BALANCE(stash, 100);
  SET-BALANCE(controller, 100);
  BOND(stash, controller, 10);
  UNBOND(controller, 10);
  time  $\leftarrow$  TIMER(WITHDRAW-UNBONDED(controller));
  ADD-TO(collection, time);

```

end

end

- CREATE-ACCOUNT(*name*, *index*)
 - Creates a Blake2 hash of the concatenated input of *name* and *index* representing the address of a account. This function only creates an address and does not conduct any I/O.
- SET-BALANCE(*account*, *balance*)
 - Sets a initial *balance* for the specified *account* in the storage state.
- BOND(*stash*, *controller*, *amount*)
 - Bonds the specified *amount* for the *stash* and *controller* pair.
- UNBOND(*account*, *amount*)
 - Unbonds the specified *amount* for the given *account*.
- WITHDRAW-UNBONDED(*controller*)
 - Withdraws the the full unbonded amount of the specified *controller* account. This represents the target Runtime function to be benchmarked

- `ADD-TO(collection, time)`
 - Adds a returned time measurement (*time*) to *collection*.
- `TIMER(function)`
 - Measures the time from the start of the specified *function* to its completion.

7 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.

7.1 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:

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

7.2 Definitions in Polkadot

The Polkadot Runtime defines the following values:

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

$$\text{weight fee} = \text{weight} \times (100 \text{ uDOTs} \div (10 \times 10^4))$$

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.

7.3 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:

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

7.3.1 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>.