Polkadot Weights

Web3 Foundation

July 2020

Contents

1 Motivation

The Polkadot network, like any other permissionless system, needs to implement a mechanism to measure and to limit the usage in order to establish an economic incentive structure, to prevent the network overload, and to mitigate DoS vulnerabilities. In particular, Polkadot enforces 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.

In contrast to some other systems such as Ethereum which implement fine measurement for each executed low-level operation by smart contracts, known as gas metering, Polkadot takes a more relaxed approach by implementing a measuring system where the cost of the transactions (referred to as 'extrinsics') are determined before execution and are known as the weight system.
The Polkadot weight system introduces a mechanism for block producers to measure the cost of running the extrinsics and determine how "heavy" it is in terms of computational cost. Within 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 2.1). Moreover, the weight system can be used to calculate a fee for executing each extrinsics according to its weight (as described in section 6.1).

Additionally, Polkadot introduces a specified block ratio (as defined in section 2.1), 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.
To begin, we introduce in Section 2 the assumption upon which the Polkadot transaction weight system is designed. In Section 2.1, we discuss the limitation Polkadot needs to enforce on the block size. In Section 3, we describe in detail the procedure upon which the weight of any transaction should be calculated. In Section 5, we present how we apply this procedure to compute the weight of particular runtime functions.

2 Assumptions

In this section, we define the concept of weight and we discuss the considerations that need to be accounted for when assigning weight to transactions. These considerations are essential in order for the weight system to deliver its fundamental mission, i.e. the fair distribution of network resources and preventing a network overload. In this regard, weights serve as an indicator on whether a block is considered full and how much space is left for remaining, pending extrinsics. Extrinsics which require too many resources are discarded. More formally, the weight system should:

- prevent the block from being filled with too many extrinsics
- avoid extrinsics where its execution takes too long, by assigning a transaction fee to each extrinsic proportional to their resource consumption.

These concepts are formalized in Definitions 1 and 4:

Definition 1 *For a block B with $Head(B)$ and $Body(B)$ the block length of B, $Len(B)$, is defined as the amount of raw bytes of B.*

Definition 2 *Targeted time per block denoted by $T(B)$ TODO: TBD.*

Definition 3 *Available block ration reserved for normal, noted by $R(B)$, is defined as the maximum weight of none-operational transactions in the Body of B divided by $Len(B)$.*

Definition 4 *Polkadot block limits as defined here should be respected by each block producer for the produced block B to be deemed valid:*

1. $Len(B) \leq 5 \times 1'024 \times 1'024 = 5'242'880$ *Bytes*

2. $T(B) = 6$ *seconds*

3. $R(B) \leq 0.75$

Definition 5 *The Polkadot transaction weight function denoted by* $\mathcal{W}$ *as follows:*

$$\mathcal{W} : \mathcal{E} \to \mathbb{N}$$
$$\mathcal{W} : E \mapsto w$$

where w *is a non-negative integer representing the weight of the extrinsic* E*. We define the weight of all inherent extrinsics as defined in [?, Definition 3.3] to be equal to 0. we extend the definition of* $\mathcal{W}$ *function to compute the weight of the block as sum of weight of all extrinsics its includes:*

$$\mathcal{W} : \mathcal{B} \to \mathbb{N}$$
$$\mathcal{W} : B \mapsto \sum_{E \in B} (W(E))$$

In the remainder of this section, we discuss the requirements to which the weight function needs to comply to.

- Computations of function $\mathcal{W}(E)$ must be determined before execution of that E.
- Due to the limited time window, computations of $\mathcal{W}$ must be done quickly and consume few resources themselves.
- $\mathcal{W}$ must be self contained and must not require I/O on the chain state. $\mathcal{W}(E)$ must depend solely on the Runtime function representing E and its parameters.

Heuristically, "heaviness" corresponds to the resources consumption of an extrinsic. In that way, the $\mathcal{W}$ value for various extrinsics should be proportional to their execution time and consumption of other system resources such as memory and amount I/O operation. For example, if Extrinsic A takes three times longer to execute than Extrinsic B while both Extrinsics consumes require similar amount of memory and I/O operations, then Extrinsic A should roughly weighs 3 times of Extrinsic B. Or

$$\mathcal{W}(A)\ 3 \times \mathcal{W}(B)$$

Nonetheless, $\mathcal{W}(E)$ can be manipulated depending on the priority of E the chain is supposed to endorse.

2.1 Limitations

In this section we discuss how applying the limitation defined in Definition 4 can be translated to limitation $\mathcal{W}$. In order to be able to translate those into concrete numbers, we need to identify an arbitrary maximum weight to which we scale all other computations. For that we first define the block weight and then assume a maximum on it block length in Definition 6:

Definition 6 *We define the block weight of block* B*, formally denoted as* $\mathcal{W}(B)$*, to be:*

$$\mathcal{W}(B) = \sum_{n=0}^{|\mathcal{E}|} (W(E_n))$$

We require that:

$$\mathcal{W}(B) < 2'000'000'000'000$$

The weights must fulfil 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 (average) weight for each transaction at about 1'500'000. Block producers have economic incentive to include as many extrinsics as possible (without exceeding limitations) into a block before reaching the targeted block time. Weights give indicators to block producers on which extrinsics to include in order to reach the blocks fullest potential.

Do note that the smallest, non-zero weight in Polkadot is set at 10'000.(why do we need this, we need to justify having a minimum)

3 Calculation of the weight function

In order to calculate weight of block B, $TWF(B)$, one needs to evaluate the weight of each transaction included in the block. Each transaction causes the execution certain Runtime functions. As such, to calculate the weight of a transaction, those functions must be analyzed in order to determine parts of the code which can significantly contribute to the execution time and consume resources such as loops, I/O operations, and data manipulation. Subsequently the performance and resource consumption of each part will be evaluated based on variety of input parameters. Based on those observations, weights are assigned Runtime functions or parameters which contribute to heavy resource consumption. These sub component of the code are discussed in Section 4.1.

The general algorithm to calculate $\mathcal{W}(E)$ is described in the Section 4.

The final assigned weights are calculated by benchmarking the Runtime functions, which is described in section 4 needs to relate the computation of final weight to the primitive otherwise the paragraph looks disconnected).

Section 5 walks through two practical examples of Runtime function analysis.

4 Benchmarking

Calculating the extrinsic weight solely based on theoretical complexity of the underlying implementation proves to be too complicated and unreliable at the same time. Certain decisions in the source code architecture, internal communication within the Runtime or other design choices could add enough overhead to make the asymptotic complexity practically meaningless.

On the other hand, benchmarking an extrinsics in a black-box fashion could (using random parameters) most centainly results in missing corner cases and worst case senarios. Instead, we benchmark all available Runtime functions which are invoked in the course of execution of extrinsics with a large collection of carefully selected input parameters and use the result of the benchmarking process to evaluate $\mathcal{W}(E)$.

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

Not every possible benchmarking outcome can be invoked by varying input parameters of the Runtime function. In some circumstances, preliminary work is required before a specific benchmark can be reliably measured, such as creating certain preexisting entries in the storage or other changes to the environment. The Practical Examples Section 5 covers this in some more detail.

4.1 Primitive Types

The Runtime reuses components, known as "primitives", to interact with the state storage. The execution cost of those primitives can be measured and a weight should be applied for each occurrence 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 mulitple values, datasets, where each entry has its corresponding, unique key.

  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. This type is also known as "child storage", where the first key is the "parent key" and the second key is the "child key". This is useful in order to scope storage entries (child keys) under a certain `context` (parent key), which is arbitrary. Therefore, one can have separated storage entries based on the context.

  The final key-value pair is stored under the key:

  ```
  hash(module_prefix) + hash(storage_prefix)
    + hash(encode(key1)) + hash(encode(key2))
  ```

It depends on the functionality of the Runtime module (or its sub-processes, rather) which storage type to use. In some cases, only a single value is required. In others, multiple values need to be fetched or inserted from/into the database.

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 or entire datasets are being worked on. Single value operations are generally quite cheap and its execution time does not vary depending on the data that's being processed. However, excessive overhead can appear when I/O operations are executed repeatedly, such as in 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, have additional overhead when inserting or retrieving datasets, which vary in sizes. 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, as described in paragraph 4.1.1.
- **Input parameters** - Input parameters that users pass on to the Runtime function can result in expensive operations. Depending on the data type, it can be appropriate to add additional weights based on certain properties, such as data size, assuming the data type allows varying sizes. The Runtime must define limits on 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 the minimum values to the maximum value, as described in paragraph 4.1.1.

What the maximum value should be really depends on the functionality that the Runtime function is trying to provide. If the choice for that value is not obvious, then it's advised to run benchmarks on a big range of values and pick a conservative value below the `targeted time per block` limit as described in section 2.1.

4.2 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 randomised 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 4.1.1.

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.

4.3 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.

4.4 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.

5 Practical examples

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

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. The analysis process shows indicators on how the preliminary work should be implemented.

5.1 Practical Example #1

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, entities elected by the Polkadot public referendum, 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 Identiy Info can contain additional fields, set manually by the corresponding account holder.

The function `request_judgement` from the `identity` pallet allows users to request judgement from a specific registrar. If this function is called by a previously judged user it implies that the Identity Info should be rejudged. Studying this function reveals multiple design choices that can impact performance, as it will be revealed by this analysis.

5.1.1 Analysis

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 all fields that have a value set, in an ordered form. It then proceeds to search all those entries for the specified registrar index. If the entry can be found, the corresponding value is updated to the value passed on as the function parameters (assuming the registrar is not "stickied", which implies it cannot be changed). If the entry cannot be found, the value is inserted into the index where a matching element can 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),
}
```

In the end, the function ensures that the sender of the extrinsic has the required balance in order to pay the fee, a balance that is adjustable and determined by the Runtime itself. Then, an event is created to insert the Identity Info into storage. The creation of events is lightweight, but its execution is what will actually commit the state changes.

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

5.1.2 Considerations

The following points must be considered:

- Varying amount of registrars.
- Varying 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.
- Varying sizes of Identity Info, including additional fields.
- It is legitimate to introduce additional weights for changes the sender has influence over, such the additional fields in the Identity Info.

5.1.3 Preliminary Work

The Polkadot Runtime specifies the `MaxRegistrars` constant, which will prevent the list of registrars of reaching an undesired length. This value should have some influence on the benchmarking process.

The benchmarking implementation of for the function *request_judgement* can be defined as follows:

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

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

```
Function Main is
    Init: collection = {};
    Pollute-Storage;
    for amount ← 1 to MaxRegistrars increment by 1 do
        Generate-Registrars(amount);
        caller ← Create-Account("caller", 1);
        Set-Balance(caller, 100);
        time ← Timer(Request-Judgement(Random(amount), 100));
        Add-To(collection, time);
    end
    return collection
end
```

- Generate-Registrars(*amount*)

  - Creates *amount* of registrars and inserts those records into storage.

- 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.

- Timer($function$)

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

- Request-Judgement(*registrar_index*, max_fee)

  - Calls the corresponding *request_judgement* Runtime function and passes on the required parameters.

- Random(*num*)

  - Picks a random number between 0 and *num*. This should be used when the benchmark should account for unpredictable values.

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

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

5.2 Practical Example #2

5.2.1 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 iterators over a range of nominators, which varies, and does I/O operation for each of them.

First, this function makes some basic checks to verify if the specified era is not higher then 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 information about the stash, such as actively bonded balance and 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()));
  }
}
```

5.2.2 Considerations

The following points must be considered:

- The Ledger contains a varying list of claimed rewards. Fetching, retaining and searching through it can affect execution time. The retained list is inserted back into storage.
- Looping through a list of nominators and creating I/O operations for each increases execution time. The Runtime fetches up to 64 nominators.

5.2.3 Preliminary Work

The Polkadot Runtime defines the `HistoryDepth` constant, which dictates the amount of Eras the reward system takes under consideration. Additionally, the constant `MaxNominatorRewardedPerValidator` specifies the maximum amount of the highest-staked nominators which will get a reward. Those values should have some influence in the benchmarking process.

The benchmarking implementation for the function *payout_stakers* can be defined as follows:

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

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

```
Function Main is
    Init: collection = {};
    Pollute-Storage();
    for amount ← 1 to MaxNominatorRewardedPerValidator increment by 1 do
        for era_depth ← 1 to HistoryDepth increment by 1 do
            validator ← Generate-Validator();
            Validate(validator);
            nominators ← Generate-Nominators(amount);
            for nominator ∈ nominators do
                Nominate(validator, nominator)
            end
            era_index ← Create-Rewards(validator, nominators, era_depth);
            time ← Timer(Payout-Stakers(validator), era_index));
            Add-To(collection, time);
        end
    end
    return collection
end
```

- Generate-Validator()
  - Creates a validators with some unbonded balances.
- Validate(*validator*)
  - Bonds balances of *validator* and bonds balances.
- Generate-Nominators(*amount*)
  - Creates the *amount* of nominators with some unbonded balances.
- Nominate(*validator*, *nominator*)
  - Starts nomination of *nominator* for *validator* by bonding balances.
- Create-Rewards(*validator*, *nominators*, *era_depth*)

- Starts an Era and creates pending rewards for *validator* and *nominators*

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

5.3 Practical Example #3

The *transfer* function of the *balances* module is designed to move the specified balance by the sender to the receiver.

5.3.1 Analysis

The source code of this function is quite short:

```
let transactor = ensure_signed(origin)?;
let dest = T::Lookup::lookup(dest)?;
<Self as Currency<_>>::transfer(
  &transactor,
  &dest,
  value,
  ExistenceRequirement::AllowDeath
)?;
```

However, what's interesting about this function, especially regarding the property `AllowDeath`, is how it manages (non-)existing accounts. Two types of behaviors are to consider:

- 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).

5.3.2 Considerations

Specific parameters can could have a big impact for this specific function. In order to trigger the two behaviors mentioned above, 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 4 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 4.

5.3.3 Preliminary Work

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

Algorithm 3: 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* ← 1 **to** 1′000 **increment by** 1 **do**
    time ← RUN-BENCHMARK(*index*, *balance*);
    ADD-TO(*collection*, *time*);
  end
  Init: *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-ACCOUNT(*"caller"*, *index*);
  recipient ← CREATE-ACCOUNT(*"recipient"*, *index*);
  SET-BALANCE(*sender*, *balance*);
  time ←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.

5.4 Practical Example #4

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. It contains some operations which have some I/O overhead.

5.4.1 Analysis

Similarly to the *payout_stakers* function (TODO: ref), this function fetches the Ledger which contains information about the stash, such as bonded balance and unlocking balance (balance that will eventually be freed and can be withdrawn).

```
if let Some(current_era) = Self::current_era() {
  ledger = ledger.consolidate_unlocked(current_era)
}
```

The function *consolidate_unlocked* does some cleaning up on the ledger, where it removes outdated entries from the unlocking balance (which implies that balance is now free and is no longer awaiting unlock).

```
let mut total = self.total;
let unlocking = self.unlocking.into_iter()
  .filter(|chunk| if chunk.era > current_era {
    true
  } else {
    total = total.saturating_sub(chunk.value);
    false
  })
  .collect();
```

This function does a check on wether the updated ledger has any balance left in regards to staking, both in terms of locked, staking balance and unlocking balance. If not amount is left, the all information related to the stash will be deleted. This results in multiple I/O calls.

```
if ledger.unlocking.is_empty() && ledger.active.is_zero() {
  // This account must have called `unbond()` with some value that caused the active
  // portion to fall below existential deposit + will have no more unlocking chunks
  // left. We can now safely remove all staking-related information.
  Self::kill_stash(&stash, num_slashing_spans)?;
  // remove the lock.
  T::Currency::remove_lock(STAKING_ID, &stash);
  // This is worst case scenario, so we use the full weight and return None
  None
```

The resulting call to $Self :: kill_stash()$ triggers:

```
slashing::clear_stash_metadata::<T>(stash, num_slashing_spans)?;
<Bonded<T>>::remove(stash);
<Ledger<T>>::remove(&controller);
<Payee<T>>::remove(stash);
<Validators<T>>::remove(stash);
<Nominators<T>>::remove(stash);
```

Alternatively, if there's some balance left, the adjusted ledger simply gets updated back into storage.

```
// This was the consequence of a partial unbond. just update the ledger and move on.
Self::update_ledger(&controller, &ledger);
```

Finally, it withdraws the unlocked balance, making it ready for transfer:

```
let value = old_total - ledger.total;
Self::deposit_event(RawEvent::Withdrawn(stash, value));
```

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 are used solely for account generation.

5.4.2 Considerations

Two important points in the *withdraw_unbonded* function must be considered. The benchmarks should trigger both conditions

- The updated ledger is inserted back into storage.
- If the stash gets killed, then multiple, repetitive deletion calls are performed in the storage.

5.4.3 Preliminary Work

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

Algorithm 4: Run multiple benchmark iterations for $withdraw_unbonded$ Runtime function

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

Function MAIN **is**
  Init: *collection* = {};
  for *balance* ← 1 **to** 100 **increment by** 1 **do**
    stash ← CREATE-ACCOUNT(*"stash"*, 1);
    controller ← CREATE-ACCOUNT(*"controller"*, 1);
    SET-BALANCE(*stash*, 100);
    SET-BALANCE(*controller*, 1);
    BOND(*stash*, *controller*, *balance*);
    PASS-ERA;
    UNBOND(*controller*, *balance*);
    PASS-ERA;
    time ←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*.

- PASS-ERA

  - Pass one era. Forces the funtion *withdraw_unbonded* to update the ledger and eventually delete information.

- 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.

6 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.

6.1 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:

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

6.2 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.

6.3 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:

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

6.3.1 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`.