Benchmarking polymorphic vs monomorphic encoding

This notebook compares two different backend version. In the traditional polymorphic one, a universal domain `$Value` is used which is the union of all possible values. Structs are represented as `Vec $Value`. For generic values, `$Value` is used, otherwise the unboxed representation wherever this is possible (non-generic parameters and locals). Equality over `$Value` is available and uses stratification to bound the recursion depth.

The monomorphic backend encoding differs as follows:

- Structs are represented as ADTs. Structs and vectors are specialized for all type instaniations found in the program. This also means that equality is specialized and does not require stratification any longer. Specification functions are specialized as well.
- Memory is specialized. We now access memory via a single address index as the type index is compiled away.
- Mutations are strongly typed as `$Mutation T`. This assumes strong edges for write-back.
- We verify a generic function (and the memory it uses) by declaring the type parameters as global given types. The conjecture here is that if verification succeeds for this, it will also succeed for every instantiation (parametric polymorphism). This probably likely needs a more formal proof down the road.
- For inlined functions, we generate specialized versions for instantiations on the call site. For calls to opaque functions, we specialize the pre and post conditions at the caller side and insert them there.

In [3]:
```
:sccache 1
:dep prover-lab = { path = "../.." }
```

Out[3]:
```
sccache: true
```

Make functions from the benchmark module available:

In [4]:
```
use prover_lab::benchmark::*;
```

Module Verification Time

This compares module by module verification wall-to-wall times, single core. Notice that functions which timeout verification are excluded.

Overall, verification times compare as follows:

In [5]:
```
let mut poly_mod = read_benchmark("poly_backend.mod_data")?;
let mut mono_mod = read_benchmark("mono_backend.mod_data")?;
stats_benchmarks(&[&poly_mod, &mono_mod])
```

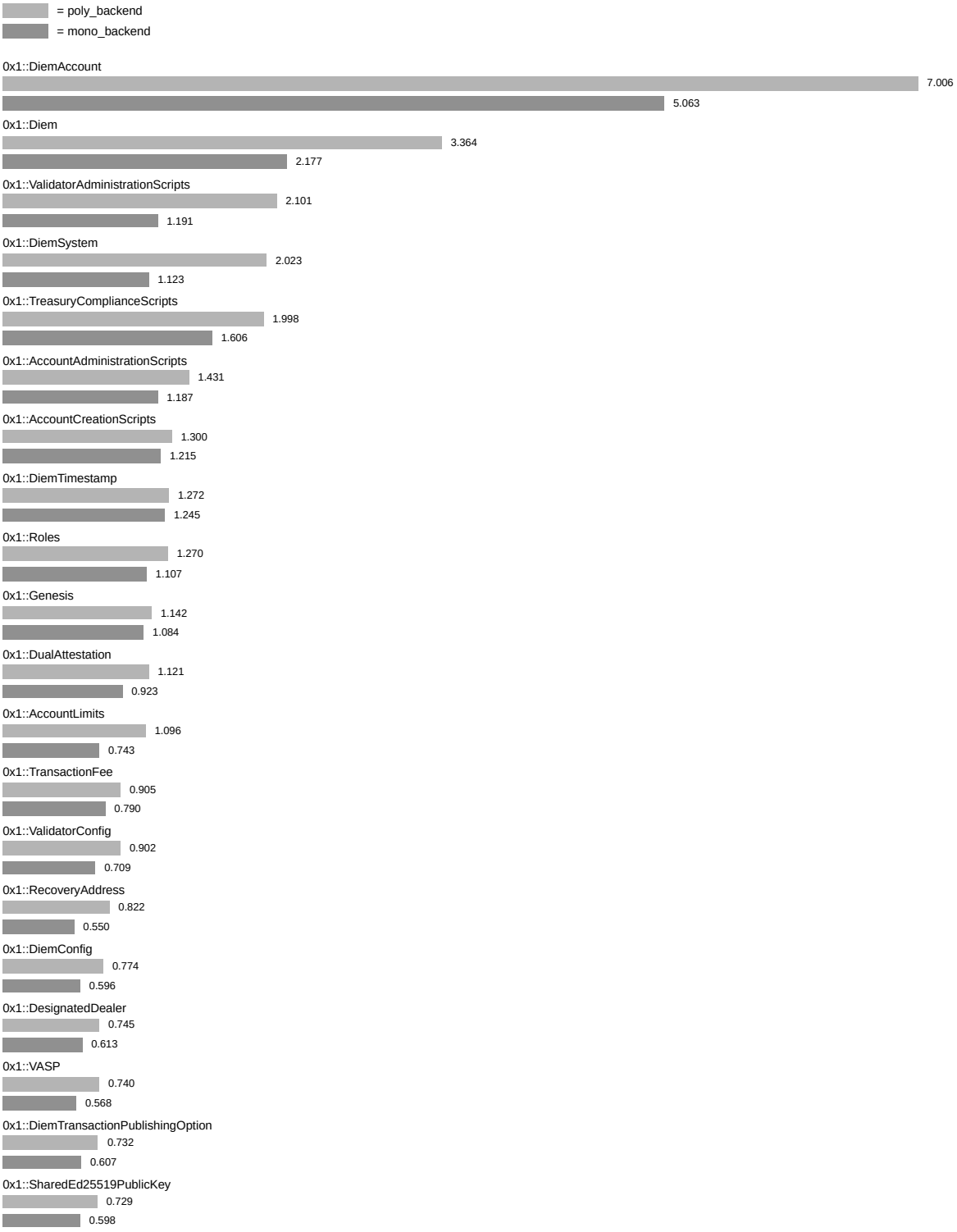
Out[5]: " poly_backend=1.000 mono_backend=0.765"

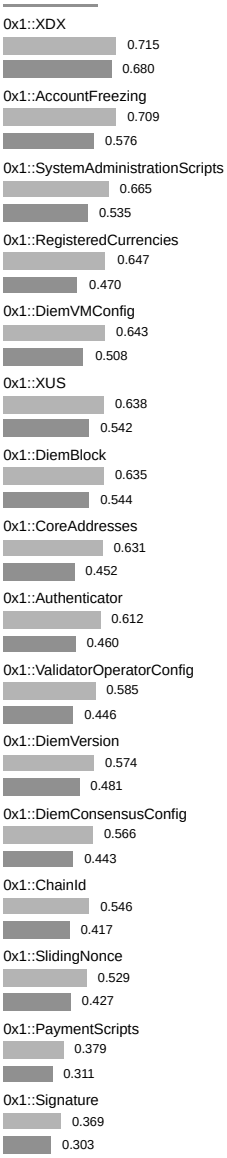
This roughly 25% performance increase is less than one might expect. However, as seen in the lab `../struct_as_adt`, it appears that `Vec $Value` is superior to ADT representations if it comes to update, which is frequent in Move programs. This advantage seems to have been overtaken by the other improvements of monomorphization.

Below are individual verification times:

In [6]:
```
poly_mod.sort(); // Will also determine order of other samples.
plot_benchmarks(&[&poly_mod, &mono_mod])
```

Out[6]:





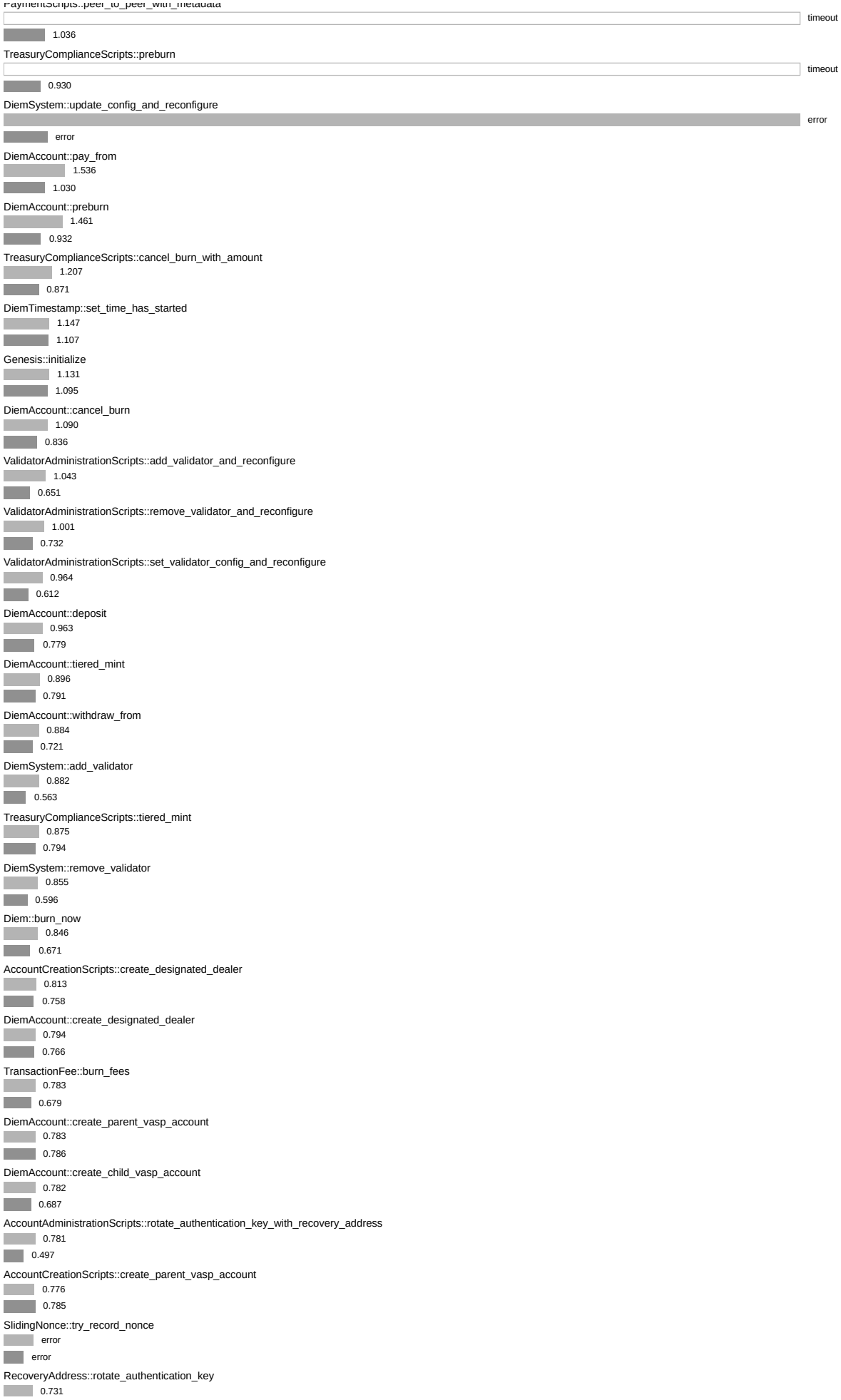
Top 30 by Function

This shows the top 30 regards verification time of individual functions. Functions which timeout or have errors *are* included here (`pragma verify = false` is ignored).

Notice that the mono backend resolves all existing timeouts. In the case of `DiemSystem::update_config_and_reconfigure` , an error is quickly produced by the mono backend while the poly backend either timed out or also produced one (as in this run).

In [7]:
```
let mut poly_fun = read_benchmark("poly_backend.fun_data")?;
let mut mono_fun = read_benchmark("mono_backend.fun_data")?;
poly_fun.sort(); // Will also determine order of other samples.
poly_fun.take(30);
plot_benchmarks(&[&poly_fun, &mono_fun])
```

Out[7]:


PaymentScripts::peer_to_peer_with_metadata
timeout
1.036
TreasuryComplianceScripts::preburn
timeout
0.930
DiemSystem::update_config_and_reconfigure
error
error
DiemAccount::pay_from
1.536
1.030
DiemAccount::preburn
1.461
0.932
TreasuryComplianceScripts::cancel_burn_with_amount
1.207
0.871
DiemTimestamp::set_time_has_started
1.147
1.107
Genesis::initialize
1.131
1.095
DiemAccount::cancel_burn
1.090
0.836
ValidatorAdministrationScripts::add_validator_and_reconfigure
1.043
0.651
ValidatorAdministrationScripts::remove_validator_and_reconfigure
1.001
0.732
ValidatorAdministrationScripts::set_validator_config_and_reconfigure
0.964
0.612
DiemAccount::deposit
0.963
0.779
DiemAccount::tiered_mint
0.896
0.791
DiemAccount::withdraw_from
0.884
0.721
DiemSystem::add_validator
0.882
0.563
TreasuryComplianceScripts::tiered_mint
0.875
0.794
DiemSystem::remove_validator
0.855
0.596
Diem::burn_now
0.846
0.671
AccountCreationScripts::create_designated_dealer
0.813
0.758
DiemAccount::create_designated_dealer
0.794
0.766
TransactionFee::burn_fees
0.783
0.679
DiemAccount::create_parent_vasp_account
0.783
0.786
DiemAccount::create_child_vasp_account
0.782
0.687
AccountAdministrationScripts::rotate_authentication_key_with_recovery_address
0.781
0.497
AccountCreationScripts::create_parent_vasp_account
0.776
0.785
SlidingNonce::try_record_nonce
error
error
RecoveryAddress::rotate_authentication_key
0.731

In [ ]: