Perun Documentation

Release 0.13

Tomas Fiedor, Jiri Pavela, et al.

Mar 27, 2018

CONTENTS:

CHAPTER

ONE

PERUN: PERFORMANCE UNDER CONTROL

1.1 What is Perun?

Have you ever encountered a sudden performance degradation and could not figure out, when and where the degradation was introduced?

Do you think that you have no idea whether the overall performance of your application is getting better or not over the time?

Is it hard for you to set performance regression testing everytime you create a new project?

Do you ever feel that you completely loose the control of the performance of your projects?

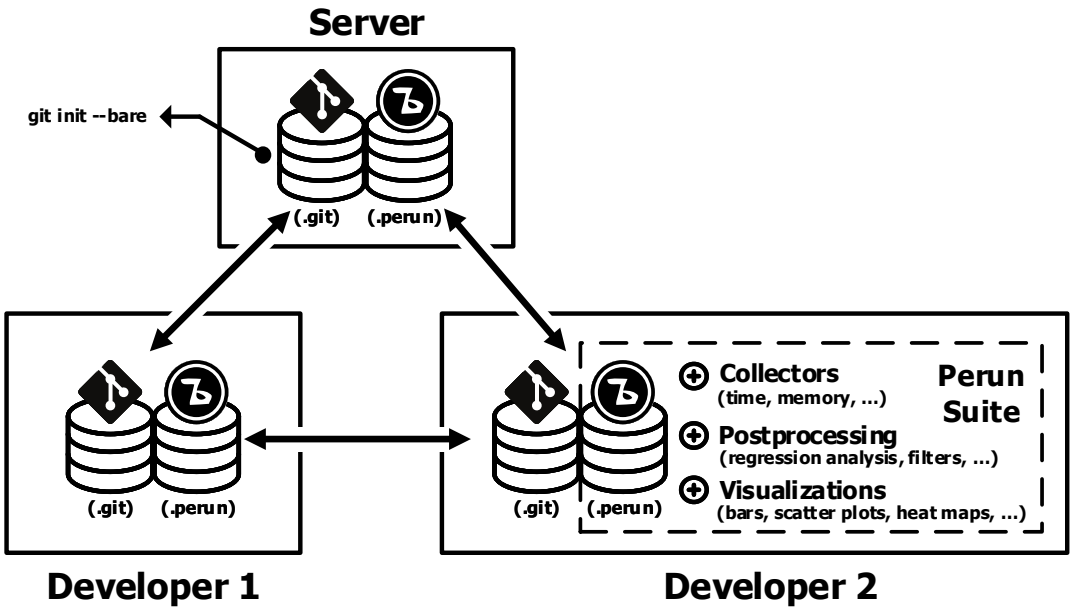
There exists solutions for managing changes of ones project—Version Control Systems (VCS)—but precise managing of performance is harder. This calls for solutions tuned to support performance management—Performance Versioning Systems.

Perun is an open source light-weight Performance Version System. While revision (or version) control systems track how your code base is changing, what features were added and keeps snapshots of versions of projects, they are mostly generic in order to satisfy needs of broad range of project types. And actually you can run all of the performance regressions tests manually and then use, e.g. git, to store the actual profiles for each minor version (e.g. commits) of your project. However, you are forced to do all of the profiling, annotations with tags and basic informations about collected resources, and many more by yourself, otherwise you lose the precise history of the performance your

application. Or you can use database, but lose the flexibility and easy usage of the versioning systems and you have to design and implement some user interface yourself.

Perun is in summary a wrapper over existing Version Systems and takes care of managing profiles for different versions of projects. Moreover, it offers a tool suite allowing one to automate the performance regression test runs, postprocess existing profiles or interpret the results. In particular, it has the following advantages over databases and sole Version Control Systems:

1. **Context**—each performance profile is assigned to a concrete minor version adding the missing context to your profiles—what was changed in the code base, when it was changed, who made the changes, etc. The profiles themselves contains collected data and addition information about the performance regression run or application configurations.
2. **Automation**—Perun allows one to easily automate the process of profile collection, eventually reducing the whole process to a single command and can be hence hooked, e.g. when one commits new changes, in supported version control system to make sure one never misses to generate new profiles for each new minor or major version of project. The specification of jobs is inspired by continuous integration systems, and is designed as YAML file, which serves as a natural format for specifying the automated jobs.
3. **Genericity**—supported format of the performance profiles is based on JSON notation and has just a minor requirements and restrictions. Perun tool suite contains a basic set of generic (and several specific) visualizations, postprocessing and collection modules which can be used as building blocks for automating jobs and interpreting results. Perun itself poses only a minor requirements for creating and registering new modules, e.g. when one wants to register new profiling data collectors, data postprocessors, customized visualiations or different version control systems.
4. **Easy to use**—the workflow, interface and storage of Perun is heavily inspired by the git systems aiming at natural use (at least for majority of potential users). Current version has a Command Line Interface consisting of commands similar to git (such as e.g. add, status, log). Interactive Graphical User Interface is currently in development.



Perun is meant to be used in two ways: (1) for a single developer (or a small team) as a complete solution for automating, storing and interpreting performance of ones project or (2) as a dedicated store for a bigger projects and

teams. Its git-like design aims at easy distribution and simple interface makes it a simple store of profiles along with the context.

Currently we are considering making a storage layer abstracting the storing of the profile either in filesystem (in git) or in database. This is currently in discussion in case the filesystem storage will not scale enough.

1.2 Installation

You can install Perun as follows:

```
make init
make install
```

These commands installs Perun to your system as a python package. You can then run perun safely from the command line using the `perun` command. Run either `perun --help` or see the *Command Line Interface* documentation for more information about running Perun commands from command line.

Note: Depending on your OS and the location of Python libraries, you might require root permissions to install Perun.

Alternatively you can install Perun in development mode:

```
make init
make dev
```

This method of installation allows you to make a changes to the code, which will be then reflected by the installation.

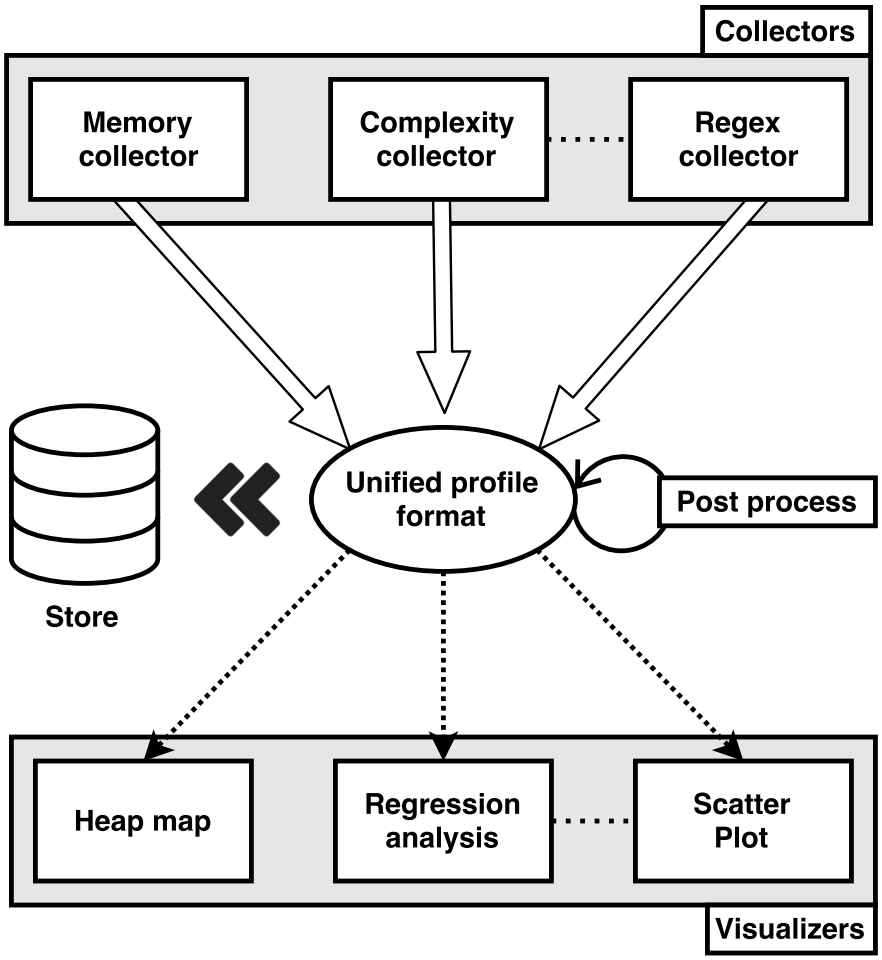
In order to partially verify that Perun runs correctly in your environment, run the automated tests as follows:

```
make test
```

In case you run in some unexpected behaviour, error or anything suspicious, either contact us directly through mail or create a new Issue.

1.3 Lifetime of a profile

Format of performance profiles is based on JSON format. It tries to unify various performance metrics and methods for collecting and postprocessing of profiling data. Profiles themselves are stored in a storage (parallel to vcs storage; currently in filesystem), compressed using the *zlib* compression method along with the additional information, such as how the profile was collected, how profiling resources were postprocessed, which metric units are used, etc. For learning how the profiles are stored in the storage and the internals of Perun refer to *Perun Internals*. For exact format of the supported profile refer to *Specification of Profile Format*.

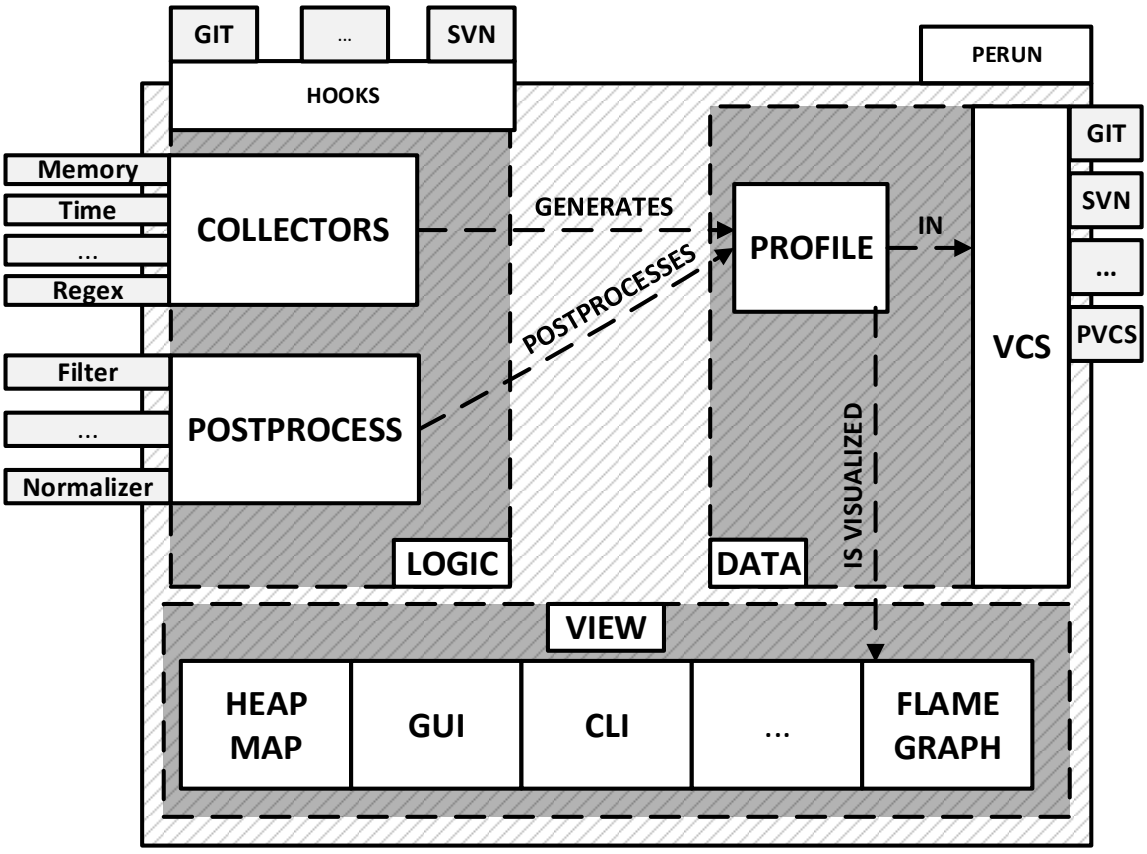


The Figure above shows the lifetime of one profile. Profiles can be generated by set of collectors (such as *Complexity Collector* which collects time durations depending on sizes of data structures, or simple *Time Collector* for basic timing) and can be further refined and transformed by sequence of postprocessing steps (like e.g. *Regression Analysis* for estimating regression models of dependent variables based on independent variables, or *Normalizer Postprocessor*, etc.).

Stored profiles then can be interpreted by set of visualization techniques like e.g. *Flame Graph*, *Scatter Plot*, or generic *Bars Plot* and *Flow Plot*. Refer to *Visualizations Overview* for more concise list and documentation of interpretation capabilities of Perun's tool suite.

1.4 Perun architecture

Internal architecture of Perun can be divided into several units—logic (commands, jobs, runners, store), data (vcs and profile), and the tool suite (collectors, postprocessors and visualizers). Data includes the core of the Perun—the profile manipulation and supported wrappers (currently git and simple custom vcs) over the existing version control systems. The logic is in charge of automation, higher-logic manipulations and takes care of actual generation of the profiles. Moreover, the whole Perun suite contains set of collectors for generation of profiles, set of postprocessors for transformation and various visualization techniques and wrappers for graphical and command line interface.



The scheme above shows the basic decomposition of Perun suite into sole units. Architecture of Perun was designed to allow simple extension of both internals and tool suite. In order to register new profiling data collector, profile postprocessor, or new visual interpretation of results refer to *Creating your own Collector*, *Examples* and *Creating your own Visualization* respectively.

1.5 List of Features

In the following, we list the foremost features and advantages of Perun:

- **Unified format**—we base our format on JSON with several minor limitations, e.g. one needs to specify header region or set of resources under fixed keys. This allows us to reuse existing postprocessors and visualisers to achieve great flexibility and easily design new methods. For full specification of our format refer to *Specification of Profile Format*.
- **Natural specification of Profiling Runs**—we base the specification of profiling jobs in Yaml format. In project configuration we let the user choose the set of collectors, set of postprocessors and configure runnable applications along with different parameter combinations and input workloads. Based on this specification we build a job matrix, which is then sequentially run and generates list of performance profiles. After the functional changes to project one then just needs to run `perun run matrix` to genereate new batch of performance profiles for latest (or currently checked-out) minor version of project.
- **Git-inspired Interface**—the *Command Line Interface* is inspired by git version control systems and specifies commands like e.g. `add`, `remove`, `status`, or `log`, well-known to basic git users. Moreover, the interface is

built using the Click library providing flexible option and argument handling. The overall interface was designed to have a natural feeling when executing the commands.

- **Efficient storage**—performance profiles are stored compressed in the storage in parallel to versions of the profiled project. Each stored object is then identified by its hash indentificator allowing quick lookup and reusing of object blobs. Storage in this form is rather packed and allows easy distribution.
- **Multiplatform-support**—Perun is implemented in Python 3 and its implementation is supported both by Windows and Unix-like platforms. However, several visualizations currently requires support for `ncurses` library (e.g. *Heap Map*).
- **Regression Analysis**—Perun's suite contains a postprocessing module for *Regression Analysis*, which supports several different strategies for finding the best model for given data (such as linear, quadratic, or constant model). Moreover, it contains switch for a more fine analysis of the data e.g. by performing regression analysis on smaller intervals, or using bisective method on whole data interval. Such analyses allows one to effectively interpret trends in data (e.g. that the duration of list search is lineary dependent on the size of the list) and help with detecting performance regressions.
- **Interactive Visualizations**—Perun's tool suite includes several visualization modules, some of them based on Bokeh visualization library, which provides nice and interactive plots, in exchange of scalability (note that we are currently exploring libraries that can scale better) —in browser, resizable and manipulable.
- **Useful API for profile manipulation**—helper modules are provided for working with our profiles in external applications (besides loading and basic usage)—we have API for executing simple queries over the resources or other parts of the profiles, or convert and transform the profiles to different representations (e.g. pandas data frame, or flame-graph format). This way, Perun can be used, e.g. together with `python` and `pandas`, as interactive interpret with support of statistical analysis.
- **Automatic Detection of Performance Degradation**—we are currently exploring effective heuristics for automatic detection of performance degradation between two project versions (e.g. between two commits). Our methodology is based on statistical methods and outputs of *Regression Analysis*. More details about degradation detection can be found at *Detecting Performance Changes*

Currently we are working on several extensions of Perun, that could be integrated in near future. Namely, in we are exploring the following possible features into Perun:

- **Regular Expression Driven Collector**—one planned collectors should be based on parsing the standard text output for a custom specified metrics, specified by regular expressions. We believe this could allow generic and quick usage to generate the performance profiles without the need of creating new specific collectors.
- **Fuzzing Collector**—other planned collector should be based on method of fuzz testing—i.e. modifying inputs in order to force error or, in our case, a performance change. We believe that this collector could generate interesting profiles and lead to a better understanding of ones applications.
- **Clustering Postprocessor**—we are exploring now how to make any profile usable for regression analysis. The notion of clustering is based on assumption, that there exists an independent variable (but unknown to us) that can be used to model the dependent variable (in our case the amount of resources). This postprocessor should try to find the optimal clustering of the dependent values in order to be usable by *Regression Analysis*.
- **Automatic Hooks**—in near future, we want to include the initially planned feature of Perun, namely the automatic hooks, that will allow to automate the runs of job matrix, automatic detection of degradation and efficient storage. Hooks would then trigger the profile collection e.g. *on_commit*, *on_push*, etc.

1.6 Overview of Customization

In order to extend the tool suite with custom modules (collectors, postprocessors and visualizations) one needs to implement `run.py` module inside the custom package stored in appropriate subdirectory (`perun.collect`, `perun.postprocess` and `perun.view` respectively). For more information about registering new profiling data collec-

tor, profile postprocessor, or new visual interpretation of results refer to *Creating your own Collector*, *Examples* and *Creating your own Visualization* respectively.

If you think your custom module could help others, please send us PR, we will review the code and in case it is suitable for wider audience, we will include it in our upstream.

1.6.1 Custom Collector

Collectors serves as a unit for generating profiles containing captured resources. In general the collection process can be broken into three phases:

1. **Before**—optional phase before the actual collection of profiling data, which is meant to prepare the profiled project for the actual collection. This phases corresponds to various initializations, custom compilations, etc.
2. **Collect**—the actual collection of profiling data, which should capture the profiled resources and ideally generate the profile w.r.t. *Specification of Profile Format*.
3. **After**—last and optional phase after resources has been successfully collected (either in raw or supported format). This phase includes e.g. corresponds filters or transformation of the profile.

Each collector should be registered in `perun.collect` package and needs to implement the proposed interfaced inside the `run.py` module. In order to register and use a new collector one needs to implement the following api in the `run.py` module:

```
def before(**kwargs):
    """(optional) Phase before execution of collector"""
    return status_code, status_msg, kwargs

def collect(**kwargs):
    """Collection of the profile---returned profile is in kwargs['profile']"""
    kwargs['profile'] = collector.do_collection()
    return status_code, status_msg, kwargs

def after(**kwargs):
    """(optional) Final postprocessing of the generated profile"""
    return status_code, status_msg, kwargs
```

For full explanation how to register and create a new collector module refer to *Creating your own Collector*.

1.6.2 Custom Postprocessor

Postprocessors in general work the same as collectors and can be broken to three phases as well. The required API to be implemented has a similar requirements and one needs to implement the following in the `run.py` module:

```
def before(**kwargs):
    """(optional) Phase before execution of postprocessor"""
    return status_code, status_msg, kwargs

def postprocess(**kwargs):
    """Postprocessing of the profile---returned profile is in kwargs['profile']"""
    kwargs['profile'] = postprocessor.do_postprocessing()
    return status_code, status_msg, kwargs

def after(**kwargs):
    """(optional) Final postprocessing of the generated profile"""
    return status_code, status_msg, kwargs
```

For full explanation how to register and create a new postprocessor module refer to *Examples*.

1.6.3 Custom Visualization

New visualizations have to be based on the *Specification of Profile Format* (or its supported conversions, see *Profile Conversions API*) and has to just implement the following in the `run.py` module:

```
import click
import perun.utils.helpers as helpers

@click.command()
@helpers.pass_profile
def visualization_name(profile, **kwargs):
    """Display the profile in custom format"""
    pass
```

The Click library is used for command line interface. For full explanation how to register and create a new collector module refer to *Creating your own Visualization*.

1.7 Acknowledgements

We thank for the support received from Red Hat (especially branch of Brno), Brno University of Technology (BUT FIT) and H2020 ECSEL project Aquas.

Further we would like to thank the following individuals (in the alphabetic order) for their (sometimes even just a little) contributions:

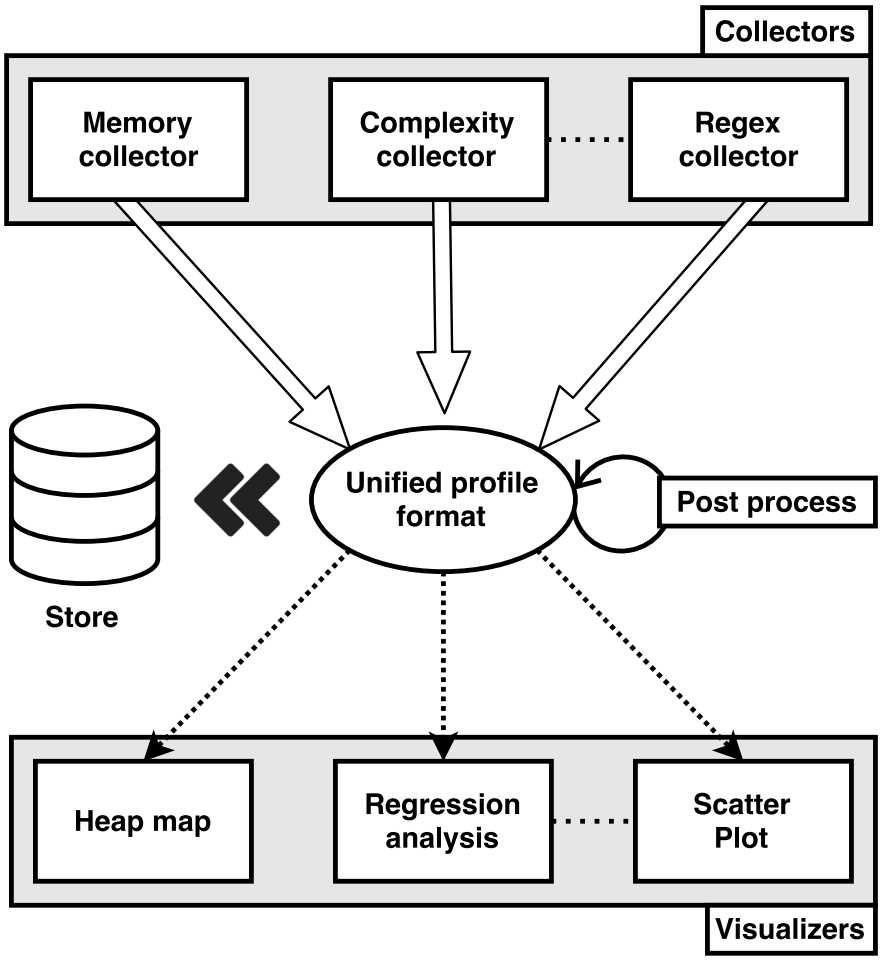
- **Jan Fiedor** (Honeywell)—for feedback, and technical discussions;
- **Martin Hruska** (BUT FIT)—for feedback, and technical discussions;
- **Petr Müller** (SAP)—for nice discussion about the project;
- **Michal Kotoun** (BUT FIT)—for feedback, and having faith in this repo;
- **Hanka Pluhackova** (BUT FIT)—for awesome logo, theoretical discussions about statistics, feedback, and lots of ideas;
- **Adam Rogalewicz** (BUT FIT)—for support, theoretical discussions, feedback;
- **Tomas Vojnar** (BUT FIT)—for support, theoretical discussions, feedback;
- **Jan Zeleny** (Red Hat)—for awesome support, and feedback.

CHAPTER

TWO

PERUN'S PROFILE FORMAT

Supported format is based on JSON with several restrictions regarding the keys (or regions) that needs to be defined inside. The intuition of JSON-like notation usage stems from its human readability and well-established support in leading programming languages (namely Python and JavaScript). Note, that however, the current version of format may generate huge profiles for some collectors, since it can contain redundancies. We are currently exploring several techniques to reduce the size of the profile.



The scheme above shows the basic lifetime of one profile. Performance profiles are generated by units called collectors (or profilers). One can either generate the profiles by its own methods or use one of the collectors from Perun's tool suite (see *Supported Collectors* for list of supported collectors). Generated profile can then be postprocessed multiple times using postprocessing units (see *Supported Postprocessors* for list of supported postprocessors), in order to e.g. normalize the values. Once you are finished with the profiles, you can store it in the persistent storage (see *Perun*

Internals for details how profiles are stored), where it will be compressed and assigned to appropriate minor version origin, e.g. concrete commit. Both stored and freshly generated profiles can be interpreted by various visualization techniques (see *Supported Visualizations* for list of visualization techniques).

2.1 Specification of Profile Format

The generic scheme of the format can be simplified in the following regions.

```
{
    "origin": "",
    "header": {},
    "collector_info": {},
    "postprocessors": [],
    "snapshots": [],
    "chunks": {}
}
```

Chunks region is currently in development, and is optional. *Snapshots* region contains the actual collected resources and can be changed through the further postprocessing phases, like e.g. by *Regression Analysis*. List of postprocessors specified in *postprocessors* region can be updated by subsequent postprocessing analyses. Finally the *origin* region is only present in non-assigned profiles. In the following we will decribe the regions in more details.

origin

```
{
    "origin": "f7f3dcea69b97f2b03c421a223a770917149cfae",
}
```

Origin specifies the concrete minor version to which the profile corresponds. This key is present only, when the profile is not yet assigned in the control system. Such profile is usually found in *.perun/jobs* directory. Before storing the profile in persistent storage, *origin* is removed and serves as validation that we are not assigning profiles to different minor versions. Assigning of profiles corresponding to different minor versions would naturally screw with the project history.

The example region above specifies, that the profile corresponded to a minor version *f7f3dc* and thus links the resources to the changes of this commit.

header

```
{
    "header": {
        "type": "time",
        "units": {
            "time": "s"
        },
        "cmd": "perun",
        "params": "status",
        "workload": "--short",
    }
}
```

Header is a key-value dictionary containing basic specification of the profile, like e.g. rough type of the performance profile, the actual command which was profiled, its parameters and input workload (giving full project configuration). The following keys are included in this region:

The example above shows header of *time* profile, with resources measured in seconds. The profiled command was `perun status --short`, which was broken down to a command `perun`, with parameter `status` and other

parameter `--short` was considered to be workload (note that the definition of workloads can vary and be used in different context).

type

Specifies rough type of the performance profile. Currently Perun consideres *time*, *mixed* and *memory*. We further plan to expand the list of choices to include e.g. *network*, *filesystem* or *utilization* profile types.

units

Map of types (and possible subtypes) of resources to their used metric units. Note that collector should guarantee that resources are unified in units. E.g. *time* can be measured in *s* or *ms*, *memory* of subtype *malloc* can be measured in *B* or *kB*, read/write thoroughput can be measured in *kB/s*, etc.

cmd

Specifies the command which was profiled and yielded the generated the profile. This can be either some script (e.g. `perun`), some command (e.g. `ls`), or execution of binary (e.g. `./out`. In general this corresponds to a profiled application. Note, that some collectors are working with their own binaries and thus do not require the command to be specified at all (like e.g. *Complexity Collector* and will thus omit the actual usage of the command), however, this key can still be used e.g. for tagging the profiles.

params

Specifies list of arguments (or parameters) for command `cmd`. This is used for more fine distinguishing of profiles regarding its parameters (e.g. when we run command with different optimizations, etc.). E.g. if take `ls` command as an example, `-al` can be considered as parameter. This key is optional, can be empty string.

workload

Similarly to parameters, workloads refer to a different inputs that are supplied to profiled command with given arguments. E.g. when one profiles text processing application, workload will refer to a concrete text files that are used to profile the application. In case of the `ls -al` command with parameters, `/` or `./subdir` can be considered as workloads. This key is optional, can be empty string.

collector_info

```
{
    "collector_info": {
        "name": "complexity",
        "params": {
            "sampling": [
                {
                    "func": "SLList_insert",
                    "sample": 1
                },
            ],
            "internal_direct_output": false,
            "internal_storage_size": 20000,
            "files": [
                "../example_sources/simple_sll_cpp/main.cpp",
                "../example_sources/simple_sll_cpp/SLList.h",
                "../example_sources/simple_sll_cpp/SLListcls.h"
            ],
            "target_dir": "./target",
            "rules": [
                "SLList_init",
                "SLList_insert",
                "SLList_search",
            ]
        },
```

```
    }
}
```

Collector info contains configuration of the collector, which was used to capture resources and generate the profile.

collector_info.name

Name of the collector (or profiler), which was used to generate the profile. This is used e.g. in displaying the list of the registered and unregistered profiles in `perun status`, in order to differentiate between profiles collected by different profilers.

collector_info.params

The configuration of the collector in the form of *(key, value)* dictionary.

The example above lists the configuration of *Complexity Collector* (for full specification of parameters refer to *Overview and Command Line Interface*). This configurations e.g. specifies, that the list of *files* will be compiled into the *target_dir* with custom Makefile and these sources will be used create a new binary for the project (prepared for profiling), which will profile function specified by *rules* w.r.t specified *sampling*.

postprocessors

```
{
    "postprocessors": [
        {
            "name": "regression_analysis",
            "params": {
                "method": "full",
                "models": [
                    "constant",
                    "linear",
                    "quadratic"
                ]
            },
        }
    ],
}
```

List of configurations of postprocessing units in order they were applied to the profile (with keys analogous to *collector_info*).

The example above specifies list with one postprocessor, namely the *Regression Analysis* (for full specification refer to *Command Line Interface*). This configuration applied regression analysis and using full *method* fully computed models for constant, linear and quadratic *models*.

snapshots

```
{
    "snapshots": [
        {
            "time": "0.025000",
            "resources": [
                {
                    "type": "memory",
                    "subtype": "malloc",
                    "address": 19284560,
                    "amount": 4,
                    "trace": [
                        {
                            "source": "../memory_collect_test.c",
```

```
                            "function": "main",
                            "line": 22
                        },
                    ],
                    "uid": {
                        "source": "../memory_collect_test.c",
                        "function": "main",
                        "line": 22
                    }
                },
            ],
            "models": []
        }, {
            "time": "0.050000",
            "resources": [
                {
                    "type": "memory",
                    "subtype": "free",
                    "address": 19284560,
                    "amount": 0,
                    "trace": [
                        {
                            "source": "../memory_collect_test.c",
                            "function": "main",
                            "line": 22
                        },
                    ],
                    "uid": {
                        "source": "../memory_collect_test.c",
                        "function": "main",
                        "line": 22
                    }
                },
            ],
            "models": []
        },
    ]
}
```


Snapshots contains the list of actual resources that were collected by the specified collector (`collector_info.name`). Each snapshot is represented by its *time*, list of captured *resources* and optionally list of *models* (refer to *Regression Analysis* for more details). The actual specification of resources varies w.r.t to used collectors.

`time`

Time specifies the timestamp of the given snapshot. The example above contains two snapshots, first captured after 0.025s and other after 0.05s of running time.

`resources`

Resources contains list of captured profiling data. Their actual format varies, and is rather flexible. In order to model the actual amount of resources, we advise to use *amount* key to quantify the size of given metric and use *type* (and possible *subtype*) in order to link resources to appropriate metric units.

The resources above were collected by *Memory Collector*, where *amount* specifies the number of bytes allocated of given memory *subtype* at given *address* by specified *trace* of functions. The first snapshot contains one resources corresponding ot *4B* of memory allocated by *malloc* in function *main* on line *22* in *memory_collect_test.c* file. The other snapshots contains record of deallocation of the given resource by *free*.

```
{
    "amount": 0.59,
    "type": "time",
    "uid": "sys"
}
```


These resources were collected by *Time Collector*, where *amount* specifies the sys time of the profile application (as obtained by `time` utility).


```
{
    "amount": 11,
    "subtype": "time delta",
    "type": "mixed",
    "uid": "SLList_init(SLList*)",
    "structure-unit-size": 0
}
```


These resources were collected by *Complexity Collector*. *Amount* here represents the difference between calling and returning the function *uid* in miliseconds, on structure of size given by *structure-unit-size*. Note that these resources are suitable for *Regression Analysis*.

models


```
{
    "uid": "SLList_insert(SLList*, int)",
    "r_square": 0.0017560012128507133,
    "coeffs": [
        {
            "value": 0.505375215875552,
            "name": "b0"
        },
        {
            "value": 9.935159839322705e-06,
            "name": "b1"
        }
    ],
    "x_interval_start": 0,
    "x_interval_end": 11892,
    "model": "linear",
    "method": "full",
}
```


Models is a list of models obtained by *Regression Analysis*. Note that the ordering of models in the list has no meaning at all. The model above corresponds to behaviour of the function `SLList_insert`, and corresponds to a linear function of $amount = b_0 + b_1 * size$ (where size corresponds to the *structure-unit-size* key of the resource) on interval $(0, 11892)$. Hence, we can estimate the complexity of function `SLList_insert` to be linear.

chunks

This region is currently in proposal. *Chunks* are meant to be a look-up table which maps unique identifiers to a larger portions of JSON regions. Since lots of informations are repeated through the profile (e.g. the *traces* in *Memory Collector*), replacing such regions with reference to the look-up table should greatly reduce the size of profiles.

2.2 Profile API

`perun.profile.factory` specifies collective interface for basic manipulation with profiles.

The format of profiles is w.r.t. *Specification of Profile Format*. This module contains helper functions for loading and storing of the profiles either in the persistent memory or in filesystem (in this case, the profile is in uncompressed format).

For further manipulations refer either to *Profile Conversions API* (implemented in `perun.profile.convert` module) or *Profile Query API* (implemented in `perun.profile.query module`). For full specification how to handle the JSON objects in Python refer to Python JSON library.

`perun.profile.factory.`**`load_profile_from_file`**(*file_name*, *is_raw_profile*)

Loads profile w.r.t *Specification of Profile Format* from file.

Parameters

- **`file_name`** (*`str`*) – file path, where the profile is stored
- **`is_raw_profile`** (*`bool`*) – if set to true, then the profile was loaded from the file system and is thus in the JSON already and does not have to be decompressed and unpacked to JSON format.

Returns JSON dictionary w.r.t. *Specification of Profile Format*

Raises **`IncorrectProfileFormatException`** – raised, when **filename** contains data, which cannot be converted to valid *Specification of Profile Format*

`perun.profile.factory.`**`store_profile_at`**(*profile*, *file_path*)

Stores profile w.r.t. *Specification of Profile Format* to output file.

Parameters

- **`profile`** (*`dict`*) – dictionary with profile w.r.t. *Specification of Profile Format*
- **`file_path`** (*`str`*) – output path, where the *profile* will be stored

2.3 Profile Conversions API

`perun.profile.convert` is a module which specifies interface for conversion of profiles from *Specification of Profile Format* to other formats.

Run the following in the Python interpreter to extend the capabilities of Python to different formats of profiles:

```
import perun.profile.convert
```

Combined with `perun.profile.factory`, `perun.profile.query` and e.g. pandas library one can obtain efficient interpreter for executing more complex queries and statistical tests over the profiles.

`perun.profile.convert.`**`resources_to_pandas_dataframe`**(*profile*)

Converts the profile (w.r.t *Specification of Profile Format*) to format supported by pandas library.

Queries through all of the resources in the *profile*, and flattens each key and value to the tabular representation. Refer to pandas libray for more possibilities how to work with the tabular representation of collected resources.

E.g. given *time* and *memory* profiles `tprof` and `mprof` respectively, one can obtain the following formats:

```
>>> convert.resources_to_pandas_dataframe(tprof)
   amount  snapshots   uid
0  0.616s          0  real
1  0.500s          0  user
2  0.125s          0   sys

>>> convert.resources_to_pandas_dataframe(mmprof)
    address  amount  snapshots subtype                   trace    type
```

```
0  19284560       4          0  malloc  malloc:unreachabl...  memory
1  19284560       0          0    free  free:unreachable:...  memory

                  uid uid:function  uid:line                uid:source
0  main:../memo...:22         main        22   ../memory_collect_test.c
1  main:../memo...:27         main        27   ../memory_collect_test.c
```

Parameters **profile** (*dict*) – dictionary with profile w.r.t. *Specification of Profile Format*

Returns converted profile to `pandas.DataFramelist` with resources flattened as a pandas dataframe

perun.profile.convert.**to_heap_map_format**(*profile*)

Simplifies the profile (w.r.t. *Specification of Profile Format*) to a representation more suitable for interpretation in the *heap map* format.

This format is used as an internal representation in the *Heap Map* visualization module. It specification is as follows:

```
{
    "type": "type of representation (heap/heat)",
    "unit": "used memory unit (string)",
    "stats": {},
    "info": [{
        "line": "(int)",
        "function": "(string)",
        "source": "(string)"
     }],
    "snapshots": [{
        "time": "time of the snapshot (string)",
        "max_amount": "maximum allocated memory in snapshot (int)",
        "min_amount": "minimum allocated memory in snapshot (int)",
        "sum_amount": "sum of allocated memory in snapshot (int)",
        "max_address": "maximal address where we allocated (int)",
        "min_address": "minimal address where we allocated (int)",
        "map": [{
            "address": "starting address of the allocation (int)",
            "amount": "amount of the allocated memory (int)",
            "uid": "index to info list with uid info (int)",
            "subtype": "allocator (string)"
        }]
    }]
}
```

Type specifies either the heap or heat representation of the data. For each *snapshot*, we have a one *map* of addresses to allocated chunks of different subtypes of allocators and *uid*. Moreover, both *snapshot* and *stats* contains several aggregated data (e.g. min, or max address) for visualization of the memory.

The usage of *Heap Map* is for visualization of address space regarding the allocations during different time's of the program (i.e. snapshots) and is meant for detecting inefficient allocations or fragmentations of memory space.

Parameters **profile** (*dict*) – profile w.r.t. *Specification of Profile Format* of **memory** *type*

Returns dictionary containing heap map representation usable for *Heap Map* visualization module.

perun.profile.convert.**to_heat_map_format**(*profile*)

Simplifies the profile (w.r.t. *Specification of Profile Format*) to a representation more suitable for interpretation in the *heat map* format.

This format is used as an internal aggregation of the allocations through all of the snapshots in the *Heap Map* visualization module. The specification is similar to `to_heap_map_format()` as follows:

```
{
    "type": "type of representation (heap/heat)",
    "unit": "used memory unit (string)",
    "stats": {
        "max_address": "maximal address in snapshot (int)",
        "min_address": "minimal address in snapshot (int)"
    },
    "map": [ ]
}
```

The main difference is in the *map*, where the data are aggregated over the snapshots represented by value representing the colours. The *warmer* the colour the more it was allocated on the concrete address.

Parameters **profile** (*dict*) – profile w.r.t. *Specification of Profile Format* of **memory** *type*

Returns dictionary containing heat map representation usable for *Heap Map* visualization module.

perun.profile.convert.**to_flame_graph_format**(*profile*)

Transforms the **memory** profile w.r.t. *Specification of Profile Format* into the format supported by perl script of Brendan Gregg.

Flame Graph can be used to visualize the inclusive consumption of resources w.r.t. the call trace of the resource. It is useful for fast detection, which point at the trace is the hotspot (or bottleneck) in the computation. Refer to *Flame Graph* for full capabilities of our Wrapper. For more information about flame graphs itself, please check Brendan Gregg's homepage.

Example of format is as follows:

```
>>> print(''.join(convert.to_flame_graph_format(memprof)))
malloc()~unreachable~0;main()~/home/user/dev/test.c~45 4
valloc()~unreachable~0;main()~/home/user/dev/test.c~75;__libc_start_main()~
↪unreachable~0;_start()~unreachable~0 8
main()~/home/user/dev/test02.c~79 156
```

Each line corresponds to some collected resource (in this case amount of allocated memory) preceeded by its trace (i.e. functions or other unique identifiers joined using `;` character.

Parameters **profile** (*dict*) – the memory profile

Returns list of lines, each representing one allocation call stack

perun.profile.convert.**plot_data_from_coefficients_of**(*model*)

Transform coefficients computed by *Regression Analysis* into dictionary of points, plotable as a function or curve. This function serves as a public wrapper over regression analysis transformation function.

Parameters **model** (*dict*) – the models dictionary from profile (refer to `models`)

Returns dict updated models dictionary extended with *plot_x* and *plot_y* lists

2.4 Profile Query API

`perun.profile.query` is a module which specifies interface for issuing queries over the profiles w.r.t *Specification of Profile Format*.

Run the following in the Python interpreter to extend the capabilities of profile to query over profiles, iterate over resources or models, etc.:

```
import perun.profile.query
```

Combined with `perun.profile.factory`, `perun.profile.convert` and e.g. Pandas library one can obtain efficient interpreter for executing more complex queries and statistical tests over the profiles.

`perun.profile.query.`**`all_resources_of`**(*profile*)

Generator for iterating through all of the resources contained in the performance profile.

Generator iterates through all of the snapshots, and subsequently yields collected resources. For more thorough description of format of resources refer to `resources`. Resources are not flattened and, thus, can contain nested dictionaries (e.g. for *traces* or *uids*).

Parameters **`profile`** (*dict*) – performance profile w.r.t *Specification of Profile Format*

Returns iterable stream of resources represented as pair `(int, dict)` of snapshot number and the resources w.r.t. the specification of the `resources`

Raises

- **`AttributeError`** – when the profile is not in the format as given by *Specification of Profile Format*
- **`KeyError`** – when the profile misses some expected key, as given by *Specification of Profile Format*

`perun.profile.query.`**`all_items_of`**(*resource*)

Generator for iterating through all of the flattened items contained inside the resource w.r.t `resources` specification.

Generator iterates through all of the items contained in the *resource* in flattened form (i.e. it does not contain nested dictionaries). Resources should be w.r.t `resources` specification.

E.g. the following resource:

```
{
    "type": "memory",
    "amount": 4,
    "uid": {
        "source": "../memory_collect_test.c",
        "function": "main",
        "line": 22
    }
}
```

yields the following stream of resources:

```
("type", "memory")
("amount", 4)
("uid", "../memory_collect_test.c:main:22")
("uid:source", "../memory_collect_test.c")
("uid:function", "main")
("uid:line": 22)
```

Parameters **`resource`** (*dict*) – dictionary representing one resource w.r.t `resources`

Returns iterable stream of `(str, value)` pairs, where the `value` is flattened to either a *string*, or *decimal* representation and `str` corresponds to the key of the item

`perun.profile.query.`**`all_resource_fields_of`**(*profile*)

Generator for iterating through all of the fields (both flattened and original) that are occuring in the resources.

Generator iterates through all of the resources and checks their flattened keys. In case some of the keys were not yet processed, they are yielded.

E.g. considering the example profiles from `resources`, the function yields the following for *memory*, *time* and *complexity* profiles respectively (considering we convert the stream to list):

```
memory_resource_fields = [
    'type', 'address', 'amount', 'uid:function', 'uid:source',
    'uid:line', 'uid', 'trace', 'subtype'
]
time_resource_fields = [
    'type', 'amount', 'uid'
]
complexity_resource_fields = [
    'type', 'amount', 'structure-unit-size', 'subtype', 'uid'
]
```

Parameters **profile** (*dict*) – performance profile w.r.t *Specification of Profile Format*

Returns iterable stream of resource field keys represented as *str*

perun.profile.query.**all_numerical_resource_fields_of**(*profile*)

Generator for iterating through all of the fields (both flattened and original) that are occuring in the resources and takes as domain integer values.

Generator iterates through all of the resources and checks their flattened keys and yields them in case they were not yet processed. If the instance of the key does not contain integer values, it is skipped.

E.g. considering the example profiles from `resources`, the function yields the following for *memory*, *time* and *complexity* profiles respectively (considering we convert the stream to list):

```
memory_num_resource_fields = ['address', 'amount', 'uid:line']
time_num_resource_fields = ['amount']
complexity_num_resource_fields = ['amount', 'structure-unit-size']
```

Parameters **profile** (*dict*) – performance profile w.r.t *Specification of Profile Format*

Returns iterable stream of resource fields key as *str*, that takes integer values

perun.profile.query.**unique_resource_values_of**(*profile*, *resource_key*)

Generator of all unique key values occurring in the resources, w.r.t. `resources` specification of resources.

Iterates through all of the values of given `resource_keys` and yields only unique values. Note that the key can contain ':' symbol indicating another level of dictionary hierarchy or '::' for specifying keys in list or set level, e.g. in case of *traces* one uses `trace::function`.

E.g. considering the example profiles from `resources`, the function yields the following for *memory*, *time* and *complexity* profiles stored in variables `mprof`, `tprof` and `cprof` respectively:

```
>>> list(query.unique_resource_values_of(mprof, 'subtype')
['malloc', 'free']
>>> list(query.unique_resource_values_of(tprof, 'amount')
[0.616, 0.500, 0.125]
>>> list(query.unique_resource_values_of(cprof, 'uid')
['SLList_init(SLList*)', 'SLList_search(SLList*, int)',
 'SLList_insert(SLList*, int)', 'SLList_destroy(SLList*)']
```

Parameters

- **profile** (*dict*) – performance profile w.r.t *Specification of Profile Format*
- **resource_key** (*str*) – the resources key identifier whose unique values will be iterated

Returns iterable stream of unique resource key values

perun.profile.query.**all_key_values_of**(*resource*, *resource_key*)

Generator of all (not essentially unique) key values in resource, w.r.t `resources` specification of resources.

Iterates through all of the values of given `resource_key` and yields every value it finds. Note that the key can contain ':' symbol indicating another level of dictionary hierarchy or '::' for specifying keys in list or set level, e.g. in case of *traces* one uses `trace::function`.

E.g. considering the example profiles from `resources` and the resources `mres` from the profile of *memory* type, we can obtain all of the values of `trace::function` key as follows:

```
>>> query.all_key_values_of(mres, 'trace::function')
['free', 'main', '__libc_start_main', '_start']
```

Note that this is mostly useful for iterating through list or nested dictionaries.

Parameters

- **resource** (*dict*) – dictionary representing one resource w.r.t `resources`
- **resource_key** (*str*) – the resources key identifier whose unique values will be iterated

Returns iterable stream of all resource key values

perun.profile.query.**all_models_of**(*profile*)

Generator of all 'models' records from the performance profile w.r.t. *Specification of Profile Format*.

Takes a profile, postprocessed by *Regression Analysis* and iterates through all of its models (for more details about models refer to `models` or *Regression Analysis*).

E.g. given some complexity profile `complexity_prof`, we can iterate its models as follows:

```
>>> gen = query.all_models_of(complexity_prof)
>>> gen.__next__()
(0, {'x_interval_start': 0, 'model': 'constant', 'method': 'full',
'coeffs': [{'name': 'b0', 'value': 0.5644496762801648}, {'name': 'b1',
'value': 0.0}], 'uid': 'SLList_insert(SLList*, int)', 'r_square': 0.0,
'x_interval_end': 11892})
>>> gen.__next__()
(1, {'x_interval_start': 0, 'model': 'exponential', 'method': 'full',
'coeffs': [{'name': 'b0', 'value': 0.9909792049684152}, {'name': 'b1',
'value': 1.000004056250301}], 'uid': 'SLList_insert(SLList*, int)',
'r_square': 0.007076437903106431, 'x_interval_end': 11892})
```

Parameters **profile** (*dict*) – performance profile w.r.t *Specification of Profile Format*

Returns iterable stream of `(int, dict)` pairs, where first yields the positional number of model and latter correponds to one 'models' record (for more details about models refer to `models` or *Regression Analysis*)

perun.profile.query.**unique_model_values_of**(*profile*, *model_key*)

Generator of all unique key values occurring in the models in the resources of given performance profile w.r.t. *Specification of Profile Format*.

Iterates through all of the values of given `resource_keys` and yields only unique values. Note that the key can contain ':' symbol indicating another level of dictionary hierarchy or '::' for specifying keys in list or set

level, e.g. in case of *traces* one uses `trace::function`. For more details about the specification of models refer to `models` or *Regression Analysis*).

E.g. given some complexity profile `complexity_prof`, we can obtain unique values of keys from *models* as follows:

```
>>> list(query.unique_model_values_of(complexity_prof, 'model')
['constant', 'exponential', 'linear', 'logarithmic', 'quadratic']
>>> list(query.unique_model_values_of(cprof, 'r_square'))
[0.0, 0.007076437903106431, 0.0017560012128507133,
 0.0008704119815403224, 0.003480627284909902, 0.001977866710139782,
 0.8391363620083871, 0.9840099999298596, 0.7283427343995424,
 0.9709120064750161, 0.9305786182556899]
```

Parameters

- **profile** (*dict*) – performance profile w.r.t *Specification of Profile Format*
- **model_key** (*str*) – key identifier from *models* for which we query its unique values

Returns iterable stream of unique model key values

CHAPTER

THREE

COMMAND LINE INTERFACE

Perun can be run from the command line (if correctly installed) using the command interface inspired by git.

The Command Line Interface is implemented using the Click library, which allows both effective definition of new commands and finer parsing of the command line arguments. The intefrace can be broken into several groups:

1. **Core commands**: namely `init`, `config`, `add`, `rm`, `status`, `log`, `run` commands (which consists of commands `run job` and `run matrix`) and `check` commands (which consists of commands `check all`, `check head` and `check profiles`). These commands automate the creation of performance profiles, detection of performance degradation and are used for management of the Perun repository. Refer to *Perun Commands* for details about commands.

2. **Collect commands**: group of `collect COLLECTOR` commands, where `COLLECTOR` stands for one of the collector of *Supported Collectors*. Each `COLLECTOR` has its own API, refer to *Collect units* for thorough description of API of individual collectors.

3. **Postprocessby commands**: group of `postprocessby POSTPROCESSOR` commands, where `POSTPROCESSOR` stands for one of the postprocessor of *Supported Postprocessors*. Each `POSTPROCESSOR` has its own API, refer to *Postprocess units* for thorough description of API of individual postprocessors.

4. **View commands**: group of `view VISUALIZATION` commands, where `VISUALIZATION` stands for one of the visualizer of *Supported Visualizations*. Each `VISUALIZATION` has its own API, refer to *Show units* for thorough description of API of individual views.

5. **Utility commands**: group of commands used for developing Perun or for maintenance of the Perun instances. Currently this group contains `create` command for faster creation of new modules.

Graphical User Interface is currently in development and hopefully will extend the flexibility of Perun's usage.

3.1 perun

Perun is an open source light-weight Performance Versioning System.

In order to initialize Perun in current directory run the following:

```
perun init
```

This initializes basic structure in `.perun` directory, together with possible reinitialization of git repository in current directory. In order to set basic configuration and define jobs for your project run the following:

```
perun config --edit
```

This opens editor and allows you to specify configuration of your project and choose set of collectors for capturing resources. See *Automating Runs* and *Perun Configuration files* for more details.

In order to generate first set of profiles for your current `HEAD` run the following:

```
perun run matrix
```

```
perun [OPTIONS] COMMAND [ARGS]...
```

Options

--no-pager
Disable paging of the long standard output (currently affects only `status` and `log` outputs). See `paging` to change the default paging strategy.

-v, --verbose
Increase the verbosity of the standard output. Verbosity is incremental, and each level increases the extent of output.

Commands

add
Links profile to concrete minor version...

check
Applies for the points of version history...

collect
Generates performance profile using selected...

config
Manages the stored local and shared...

init
Initializes performance versioning system at...

log
Shows history of versions and associated...

postprocessby
Postprocesses the given stored or pending...

rm
Unlinks the profile from the given minor...

run
Generates batch of profiles w.r.t.

show
Interprets the given profile using the...

status
Shows the status of vcs, associated profiles...

utils
Contains set of developer commands, wrappers...

3.2 Perun Commands

3.2.1 perun init

Initializes performance versioning system at the destination path.

`perun init` command initializes the perun's infrastructure with basic file and directory structure inside the `.perun` directory. Refer to *Perun Internals* for more details about storage of Perun. By default following directories are created:

1. `.perun/jobs`: storage of performance profiles not yet assigned to concrete minor versions.
2. `.perun/objects`: storage of packed contents of performance profiles and additional informations about minor version of wrapped vcs system.
3. `.perun/cache`: fast access cache of selected latest unpacked profiles
4. `.perun/local.yml`: local configuration, storing specification of wrapped repository, jobs configuration, etc. Refer to *Perun Configuration files* for more details.

The infrastructure is initialized at <path>. If no <path> is given, then current working directory is used instead. In case there already exists a performance versioning system, the infrastructure is only reinitialized.

By default, a control system is initialized as well. This can be changed by by setting the `--vcs-type` parameter (currently we support `git` and `tagit`—a lightweight git-based wrapped based on tags). Additional parameters can be passed to the wrapped control system initialization using the `--vcs-params`.

```
perun init [OPTIONS] <path>
```

Options

`--vcs-type` `<vcs_type>`
: In parallel to initialization of Perun, initialize the vcs of <type> as well (by default `git`).

`--vcs-path` `<vcs_path>`
: Sets the destination of wrapped vcs initialization at <path>.

`--vcs-param` `<vcs_param>`
: Passes additional (key, value) parameter to initialization of version control system, e.g. `separate-git-dir dir`.

`--vcs-flag` `<vcs_flag>`
: Passes additional flag to a initialization of version control system, e.g. `bare`.

`-c, --configure`
: After successful initialization of both systems, opens the local configuration using the `editor` set in shared config.

Arguments

`<path>`
: Optional argument

3.2.2 perun config

Manages the stored local and shared configuration.

Perun supports two external configurations:

1. `local.yml`: the local configuration stored in `.perun` directory, containing the keys such as specification of wrapped repository or job matrix used for quick generation of profiles (run `perun run matrix --help` or refer to *Automating Runs* for information how to construct the job matrix).
2. `shared.yml`: the global configuration shared by all perun instances, containing shared keys, such as text editor, formatting string, etc.

The syntax of the `<key>` in most operations consists of section separated by dots, e.g. `vcs.type` specifies `type` key in `vcs` section. The lookup of the `<key>` can be performed in three modes, `--local`, `--shared` and `--nearest`, locating or setting the `<key>` in local, shared or nearest configuration respectively (e.g. when one is trying to get some key, there may be nested perun instances that do not contain the given key). By default, perun operates in the nearest config mode.

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options.

E.g. using the following one can retrieve the type of the nearest perun instance wrapper:

```
$ perun config get vsc.type
vcs.type: git
```

```
perun config [OPTIONS] COMMAND [ARGS]...
```

Options

-l, --local
Will lookup or set in the local config i.e. `.perun/local.yml`.

-h, --shared
Will lookup or set in the shared config i.e. `shared.yml`.

-n, --nearest
Will recursively discover the nearest suitable config. The lookup strategy can differ for `set` and `get`/`edit`.

Commands

edit
Edits the configuration file in the external...

get
Looks up the given `<key>` within the...

set
Sets the value of the `<key>` to the given...

3.2.3 perun config get

Looks up the given `<key>` within the configuration hierarchy and returns the stored value.

The syntax of the `<key>` consists of section separated by dots, e.g. `vcs.type` specifies `type` key in `vcs` section. The lookup of the `<key>` can be performed in three modes, `--local`, `--shared` and `--nearest`, locating the `<key>` in local, shared or nearest configuration respectively (e.g. when one is trying to get some key, there may be nested perun instances that do not contain the given key). By default, perun operates in the nearest config mode.

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options.

E.g. using the following can retrieve the type of the nearest perun wrapper:

```
$ perun config get vsc.type
vcs.type: git

$ perun config --shared get general.editor
general.editor: vim
```

```
perun config get [OPTIONS] <key>
```

Arguments

<key>
Required argument

3.2.4 perun config set

Sets the value of the <key> to the given <value> in the target configuration file.

The syntax of the <key> corresponds of section separated by dots, e.g. vcs.type specifies type key in vcs section. Perun sets the <key> in three modes, --local, --shared and --nearest, which sets the <key> in local, shared or nearest configuration respectively (e.g. when one is trying to get some key, there may be nested perun instances that do not contain the given key). By default, perun will operate in the nearest config mode.

The <value> is arbitrary depending on the key.

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options and their values.

E.g. using the following can set the log format for nearest perun instance wrapper:

```
$ perun config set format.shortlog "| %origin% | %collector% |"
format.shortlog: | %origin% | %collector% |
```

```
perun config set [OPTIONS] <key> <value>
```

Arguments

<key>
Required argument

<value>
Required argument

3.2.5 perun config edit

Edits the configuration file in the external editor.

The used editor is specified by the *general.editor* option, specified in the nearest perun configuration..

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options.

```
perun config edit [OPTIONS]
```

3.2.6 perun add

Links profile to concrete minor version storing its content in the `.perun` dir and registering the profile in internal minor version index.

In order to link <profile> to given minor version <hash> the following steps are executed:

1. We check in <profile> that its *origin* key corresponds to <hash>. This serves as a check, that we do not assign profiles to different minor versions.
2. The *origin* is removed and contents of <profile> are compresed using *zlib* compression method.
3. Binary header for the profile is constructed.
4. Compressed contents are appended to header, and this blob is stored in `.perun/objects` directory.
5. New blob is registered in <hash> minor version's index.
6. Unless `--keep-profile` is set. The original profile is deleted.

If no <*hash*> is specified, then current `HEAD` of the wrapped version control system is used instead. Massaging of <hash> is taken care of by underlying version control system (e.g. git uses `git rev-parse`).

<profile> can either be a `pending tag` or a fullpath. `Pending tags` are in form of `i@p`, where `i` stands for an index in the pending profile directory (i.e. `.perun/jobs`) and `@p` is literal suffix. Run `perun status` to see the *tag* anotation of pending profiles.

Example of adding profiles:

```
$ perun add mybin-memory-input.txt-2017-03-01-16-11-04.perf
```

This command adds the profile collected by *memory* collector during profiling `mybin` command with `input.txt` workload on 1st March at 16:11 to the current `HEAD`.

An error is raised if the command is executed outside of range of any perun, if <profile> points to incorrect profile (i.e. not w.r.t. *Specification of Profile Format*) or <hash> does not point to valid minor version ref.

See *Perun Internals* for information how perun handles profiles internally.

```
perun add [OPTIONS] <profile>
```

Options

-m, --minor <minor>
: <profile> will be stored at this minor version (default is HEAD).

--keep-profile
: Keeps the profile in filesystem after registering it in Perun storage. Otherwise it is deleted.

Arguments

<profile>
: Required argument(s)

3.2.7 perun rm

Unlinks the profile from the given minor version, keeping the contents stored in `.perun` directory.

<profile> is unlinked in the following steps:

1. <profile> is looked up in the <hash> minor version's internal index.
2. In case <profile> is not found. An error is raised.
3. Otherwise, the record corresponding to <hash> is erased. However, the original blob is kept in `.perun/objects`.

If no *<hash>* is specified, then current `HEAD` of the wrapped version control system is used instead. Massaging of <hash> is taken care of by underlying version control system (e.g. git uses `git rev-parse`).

<profile> can either be a `index tag` or a path specifying the profile. `Index tags` are in form of `i@i`, where `i` stands for an index in the minor version's index and `@i` is literal suffix. Run `perun status` to see the *tags* of current `HEAD`'s index.

Examples of removing profiles:

```
$ perun rm 2@i
```

This commands removes the third (we index from zero) profile in the index of registered profiles of current `HEAD`.

An error is raised if the command is executed outside of range of any Perun or if <profile> is not found inside the <hash> index.

See *Perun Internals* for information how perun handles profiles internally.

```
perun rm [OPTIONS] <profile>
```

Options

-m, --minor <minor>
: <profile> will be stored at this minor version (default is HEAD).

-A, --remove-all
: Removes all occurrences of <profile> from the <hash> index.

Arguments

<profile>
: Required argument(s)

3.2.8 perun status

Shows the status of vcs, associated profiles and perun.

Shows the status of both the nearest perun and wrapped version control system. For vcs this outputs e.g. the current minor version `HEAD`, current major version and description of the `HEAD`. Moreover `status` prints the lists of tracked and pending (found in `.perun/jobs`) profiles lexicographically sorted along with additional information such as their types and creation times.

Unless `perun --no-pager status` is issued as command, or appropriate `paging` option is set, the outputs of status will be paged (by default using `less`.

An error is raised if the command is executed outside of range of any perun, or configuration misses certain configuration keys (namely `format.status`).

Refer to *Customizing Statuses* for information how to customize the outputs of `status` or how to set *`format.status`* in nearest configuration.

```
perun status [OPTIONS]
```

Options

`-s, --short`
Shortens the output of `status` to include only most necessary informations.

3.2.9 perun log

Shows history of versions and associated profiles.

Shows the history of the wrapped version control system and all of the associated profiles starting from the <hash> point, outputing the information about number of profiles, about descriptions ofconcrete minor versions, their parents, parents etc.

If `perun log --short` is issued, the shorter version of the `log` is outputted.

In no <hash> is given, then HEAD of the version control system is used as a starting point.

Unless `perun --no-pager log` is issued as command, or appropriate `paging` option is set, the outputs of log will be paged (by default using `less`.

Refer to *Customizing Logs* for information how to customize the outputs of `log` or how to set *`format.log`* in nearest configuration.

```
perun log [OPTIONS] <hash>
```

Options

`--count-only`
Shows only aggregated data without minor version history description

`--show-aggregate`
Includes the aggregated values for each minor version.

`--last` `<last>`
Limits the output of log to last <int> entries.

`--no-merged`
Skips merges during the iteration of the project history.

`-s, --short`
Shortens the output of `log` to include only most necessary information.

Arguments

`<hash>`
Optional argument

3.2.10 perun run

Generates batch of profiles w.r.t. specification of list of jobs.

Either runs the job matrix stored in local.yml configuration or lets the user construct the job run using the set of parameters.

```
perun run [OPTIONS] COMMAND [ARGS]...
```

Options

-ot, --output-filename-template <output_filename_template>
: Specifies the template for automatic generation of output filename This way the file with collected data will have a resulting filename w.r.t to this parameter. Refer to *format.output_profile_template* for more details about the format of the template.

Commands

job
: Run specified batch of perun jobs to generate...

matrix
: Runs the jobs matrix specified in the...

3.2.11 perun run job

Run specified batch of perun jobs to generate profiles.

This command correspond to running one isolated batch of profiling jobs, outside of regular profilings. Run `perun run matrix`, after specifying job matrix in local configuration to automate regular profilings of your project. After the batch is generated, each profile is taged with *origin* set to current `HEAD`. This serves as check to not assing such profiles to different minor versions.

By default the profiles computed by this batch job are stored inside the `.perun/jobs/` directory as a files in form of:

```
bin-collector-workload-timestamp.perf
```

In order to store generated profiles run the following, with `i@p` corresponding to *pending tag*, which can be obtained by running `perun status`:

```
perun add i@p
```

```
perun run job -c time -b ./mybin -w file.in -w file2.in -p normalizer
```

This command profiles two commands `./mybin file.in` and `./mybin file2.in` and collects the profiling data using the *Time Collector*. The profiles are afterwards normalized with the *Normalizer Postprocessor*.

```
perun run job -c complexity -b ./mybin -w sll.cpp -cp complexity targetdir=./src
```

This commands runs one job './mybin sll.cpp' using the *Complexity Collector*, which uses custom binaries targeted at `./src` directory.

```
perun run job -c mcollect -b ./mybin -b ./otherbin -w input.txt -p normalizer -p␣
↪regression_analysis
```

This commands runs two jobs `./mybin input.txt` and `./otherbin input.txt` collects the profiles using the *Memory Collector*. The profiles are afterwards postprocessed, first using the *Normalizer Postprocessor* and then with *Regression Analysis*.

Refer to *Automating Runs* and *Perun's Profile Format* for more details about automation and lifetimes of profiles. For list of available collectors and postprocessors refer to *Supported Collectors* and *Supported Postprocessors* respectively.

```
perun run job [OPTIONS]
```

Options

-b, --cmd <cmd>
Command that is being profiled. Either corresponds to some script, binary or command, e.g. `./mybin` or `perun`. [required]

-a, --args <args>
Additional parameters for <cmd>. E.g. `status` or `-al` is command parameter.

-w, --workload <workload>
Inputs for <cmd>. E.g. `./subdir` is possible workloadfor `ls` command.

-c, --collector <collector>
Profiler used for collection of profiling data for the given <cmd> [required]

-cp, --collector-params <collector_params>
Additional parameters for the <collector> read from the file in YAML format

-p, --postprocessor <postprocessor>
After each collection of data will run <postprocessor> to postprocess the collected resources.

-pp, --postprocessor-params <postprocessor_params>
Additional parameters for the <postprocessor> read from the file in YAML format

3.2.12 perun run matrix

Runs the jobs matrix specified in the local.yml configuration.

This commands loads the jobs configuration from local configuration, builds the *job matrix* and subsequently runs the jobs collecting list of profiles. Each profile is then stored in `.perun/jobs` directory and moreover is annotated using by setting `origin` key to current `HEAD`. This serves as check to not assing such profiles to different minor versions.

The job matrix is defined in the yaml format and consists of specification of binaries with corresponding arguments, workloads, supported collectors of profiling data and postprocessors that alter the collected profiles.

Refer to *Automating Runs* and *Job Matrix Format* for more details how to specify the job matrix inside local configuration and to *Perun Configuration files* how to work with Perun's configuration files.

```
perun run matrix [OPTIONS]
```

3.2.13 perun check

Applies for the points of version history checks for possible performance changes.

This command group either runs the checks for one point of history (`perun check head`) or for the whole history (`perun check all`). For each minor version (called the *target*) we iterate over all of the registered profiles and try to find a predecessor minor version (called the *baseline*) with profile of the same configuration (by configuration we mean the tuple of collector, postprocessors, command, arguments and workloads) and run the checks according to the rules set in the configurations.

The rules are specified as an ordered list in the configuration by *degradation.strategies*, where the keys correspond to the configuration (or the type) and key *method* specifies the actual method used for checking for performance changes. The applied methods can then be either specified by the full name or by its short string consisting of all first letter of the function name.

The example of configuration snippet that sets rules and strategies for one project can be as follows:

degradation: apply: first strategies:

- type: mixed postprocessor: regression_analysis method: bmoe
- cmd: mybin type: memory method: bmoe
- method: aat

Currently we support the following methods:

1. Best Model Order Equality (BMOE) 2. Average Amount Threshold (AAT)

```
perun check [OPTIONS] COMMAND [ARGS]...
```

Options

-c, --compute-missing
whenever there are missing profiles in the given point of history the matrix will be rerun and new generated profiles assigned.

Commands

all
Checks for changes in performance for the...

head
Checks for changes in performance between...

profiles
Checks for changes in performance between two...

3.2.14 perun check head

Checks for changes in performance between between specified minor version (or current *head*) and its predecessor minor versions.

The command iterates over all of the registered profiles of the specified *minor version* (*target*; e.g. the *head*), and tries to find the nearest predecessor minor version (*baseline*), where the profile with the same configuration as the tested target profile exists. When it finds such a pair, it runs the check according to the strategies set in the configuration (see *Configuring Degradation Detection* or *Perun Configuration files*).

By default the `hash` corresponds to the *head* of the current project.

```
perun check head [OPTIONS] <hash>
```

Arguments

<hash>
Optional argument

3.2.15 perun check all

Checks for changes in performance for the specified interval of version history.

The commands crawls through the whole history of project versions starting from the specified `<hash>` and for all of the registered profiles (corresponding to some *target* minor version) tries to find a suitable predecessor profile (corresponding to some *baseline* minor version) and runs the performance check according to the set of strategies set in the configuration (see *Configuring Degradation Detection* or *Perun Configuration files*).

```
perun check all [OPTIONS] <hash>
```

Arguments

<hash>
Optional argument

3.2.16 perun check profiles

Checks for changes in performance between two profiles.

The commands checks for the changes between two isolate profiles, that can be stored in pending profiles, registered in index, or be simply stored in filesystem. Then for the pair of profiles <baseline> and <target> the command runs the performance chekc according to the set of strategies set in the configuration (see *Configuring Degradation Detection* or *Perun Configuration files*).

<baseline> and <target> profiles will be looked up in the following steps:

1. If profile is in form `i@i` (i.e, an *index tag*), then *ith* record registered in the minor version <hash> index will be used.
2. If profile is in form `i@p` (i.e., an *pending tag*), then *ith* profile stored in `.perun/jobs` will be used.
3. Profile is looked-up within the minor version <hash> index for a match. In case the <profile> is registered there, it will be used.
4. Profile is looked-up within the `.perun/jobs` directory. In case there is a match, the found profile will be used.
5. Otherwise, the directory is walked for any match. Each found match is asked for confirmation by user.

```
perun check profiles [OPTIONS] <baseline> <target>
```

Options

-m, --minor <minor>
Will check the index of different minor version <hash> during the profile lookup.

Arguments

<baseline>
Required argument

<target>
Required argument

3.3 Collect Commands

3.3.1 perun collect

Generates performance profile using selected collector.

Runs the single collector unit (registered in Perun) on given profiled command (optionaly with given arguments and workloads) and generates performance profile. The generated profile is then stored in `.perun/jobs/` directory as a file, by default with filename in form of:

```
bin-collector-workload-timestamp.perf
```

Generated profiles will not be postprocessed in any way. Consult `perun postprocessby --help` in order to postprocess the resulting profile.

The configuration of collector can be specified in external YAML file given by the `-p`/`--params` argument.

For a thorough list and description of supported collectors refer to *Supported Collectors*. For a more subtle running of profiling jobs and more complex configuration consult either `perun run matrix --help` or `perun run job --help`.

```
perun collect [OPTIONS] COMMAND [ARGS]...
```

Options

-c, --cmd <cmd>
Command that is being profiled. Either corresponds to some script, binary or command, e.g. `./mybin` or `perun`.

-a, --args <args>
Additional parameters for <cmd>. E.g. `status` or `-al` is command parameter.

-w, --workload <workload>
Inputs for <cmd>. E.g. `./subdir` is possible workloadfor `ls` command.

-p, --params <params>
Additional parameters for called collector read from file in YAML format.

-ot, --output-filename-template <output_filename_template>
Specifies the template for automatic generation of output filename This way the file with collected data will

have a resulting filename w.r.t to this parameter. Refer to *format.output_profile_template* for more details about the format of the template.

3.3.2 Collect units

perun collect complexity

Generates *complexity* performance profile, capturing running times of function depending on underlying structural sizes.

- **Limitations**: C/C++ binaries
- **Metric**: *mixed* (captures both *time* and *size* consumption)
- **Dependencies**: `SystemTap` (+ corresponding requirements e.g. kernel -dbgsym version)
- **Default units**: *us* for *time*, *element number* for *size*

Example of collected resources is as follows:

```
{
    "amount": 11,
    "subtype": "time delta",
    "type": "mixed",
    "uid": "SLList_init(SLList*)",
    "structure-unit-size": 0
}
```

Complexity collector provides various collection *strategies* which are supposed to provide sensible default settings for collection. This allows the user to choose suitable collection method without the need of detailed rules / sampling specification. Currently supported strategies are:

- **userspace**: This strategy traces all userspace functions / code blocks without

the use of sampling. Note that this strategy might be resource-intensive. * **all**: This strategy traces all userspace + library + kernel functions / code blocks that are present in the traced binary without the use of sampling. Note that this strategy might be very resource-intensive. * **u_sampled**: Sampled version of the **userspace** strategy. This method uses sampling to reduce the overhead and resources consumption. * **a_sampled**: Sampled version of the **all** strategy. Its goal is to reduce the overhead and resources consumption of the **all** method. * **custom**: User-specified strategy. Requires the user to specify rules and sampling manually.

Note that manually specified parameters have higher priority than strategy specification and it is thus possible to override concrete rules / sampling by the user.

Complexity profiles are suitable for postprocessing by *Regression Analysis* since they capture dependency of time consumption depending on the size of the structure. This allows one to model the estimation of complexity of individual functions.

Scatter plots are suitable visualization for profiles collected by *complexity* collector, which plots individual points along with regression models (if the profile was postprocessed by regression analysis). Run `perun show scatter --help` or refer to *Scatter Plot* for more information about *scatter plots*.

Refer to *Complexity Collector* for more thorough description and examples of *complexity* collector.

```
perun collect complexity [OPTIONS]
```

Options

-m, --method <method>
Select strategy for probing the binary. See documentation for detailed explanation for each strategy. [required]

-r, --rules <rules>
Set the probe points for profiling.

-s, --sampling <sampling>
Set the runtime sampling of the given probe points.

-g, --global_sampling <global_sampling>
Set the global sample for all probes, –sampling parameter for specific rules have higher priority.

perun collect memory

Generates *memory* performance profiel, capturing memory allocations of different types along with target address and full call trace.

- **Limitations**: C/C++ binaries
- **Metric**: *memory*
- **Dependencies**: `libunwind.so` and custom `libmalloc.so`
- **Default units**: *B* for *memory*

The following snippet shows the example of resources collected by *memory* profiler. It captures allocations done by functions with more detailed description, such as the type of allocation, trace, etc.

```
{
    "type": "memory",
    "subtype": "malloc",
    "address": 19284560,
    "amount": 4,
    "trace": [
        {
            "source": "../memory_collect_test.c",
            "function": "main",
            "line": 22
        },
    ],
    "uid": {
        "source": "../memory_collect_test.c",
        "function": "main",
        "line": 22
    }
},
```

Memory profiles can be efficiently interpreted using *Heap Map* technique (together with its *heat* mode), which shows memory allocations (by functions) in memory address map.

Refer to *Memory Collector* for more thorough description and examples of *memory* collector.

```
perun collect memory [OPTIONS]
```

Options

-s, --sampling <sampling>
Sets the sampling interval for profiling the allocations. I.e. memory snapshots will be collected each <sampling> seconds.

--no-source <no_source>
Will exclude allocations done from <no_source> file during the profiling.

--no-func <no_func>
Will exclude allocations done by <no func> function during the profiling.

-a, --all
Will record the full trace for each allocation, i.e. it will include all allocators and even unreachable records.

perun collect time

Generates *time* performance profile, capturing overall running times of the profiled command.

- **Limitations**: *none*
- **Metric**: running *time*
- **Dependencies**: *none*
- **Default units**: *s*

This is a wrapper over the `time` linux unitility and captures resources in the following form:

```
{
    "amount": 0.59,
    "type": "time",
    "uid": "sys"
}
```

Refer to *Time Collector* for more thorough description and examples of *complexity* collector.

```
perun collect time [OPTIONS]
```

3.4 Postprocess Commands

3.4.1 perun postprocessby

Postprocesses the given stored or pending profile using selected postprocessor.

Runs the single postprocessor unit on given looked-up profile. The postprocessed file will be then stored in `.perun/jobs/` directory as a file, by default with filanem in form of:

```
bin-collector-workload-timestamp.perf
```

The postprocessed <profile> will be looked up in the following steps:

1. If <profile> is in form `i@i` (i.e, an *index tag*), then *ith* record registered in the minor version <hash> index will be postprocessed.
2. If <profile> is in form `i@p` (i.e., an *pending tag*), then *ith* profile stored in `.perun/jobs` will be postprocessed.

3. <profile> is looked-up within the minor version <hash> index for a match. In case the <profile> is registered there, it will be postprocessed.
4. <profile> is looked-up within the `.perun/jobs` directory. In case there is a match, the found profile will be postprocessed.
5. Otherwise, the directory is walked for any match. Each found match is asked for confirmation by user.

For checking the associated *tags* to profiles run `perun status`.

Example 1. The following command will postprocess the given profile stored at given path by normalizer, i.e. for each snapshot, the resources will be normalized to the interval <0, 1>:

```
perun postprocessby ./echo-time-hello-2017-04-02-13-13-34-12.perf normalizer
```

Example 2. The following command will postprocess the second profile stored in index of commit preceeding the current head using interval regression analysis:

```
perun postprocessby -m HEAD~1 1@i regression_analysis --method=interval
```

For a thorough list and description of supported postprocessors refer to *Supported Postprocessors*. For a more subtle running of profiling jobs and more complex configuration consult either `perun run matrix --help` or `perun run job --help`.

```
perun postprocessby [OPTIONS] <profile> COMMAND [ARGS]...
```

Options

-ot, --output-filename-template <output_filename_template>
: Specifies the template for automatic generation of output filename This way the postprocessed file will have a resulting filename w.r.t to this parameter. Refer to *format.output_profile_template* for more details about the format of the template.

-m, --minor <minor>
: Will check the index of different minor version <hash> during the profile lookup

Arguments

<profile>
: Required argument

3.4.2 Postprocess units

perun postprocessby normalizer

Normalizes performance profile into flat interval.

- **Limitations**: *none*
- **Dependencies**: *none*

Normalizer is a postprocessor, which iterates through all of the snapshots and normalizes the resources of same type to interval `(0, 1)`, where `1` corresponds to the maximal value of the given type.

Consider the following list of resources for one snapshot generated by *Time Collector*:

```
[
    {
        'amount': 0.59,
        'uid': 'sys'
    }, {
        'amount': 0.32,
        'uid': 'user'
    }, {
        'amount': 2.32,
        'uid': 'real'
    }
]
```

Normalizer yields the following set of resources:

```
[
    {
        'amount': 0.2543103448275862,
        'uid': 'sys'
    }, {
        'amount': 0.13793103448275865,
        'uid': 'user'
    }, {
        'amount': 1.0,
        'uid': 'real'
    }
]
```

Refer to *Normalizer Postprocessor* for more thorough description and examples of *normalizer* postprocessor.

```
perun postprocessby normalizer [OPTIONS]
```

perun postprocessby regression_analysis

Finds fitting regression models to estimate models of profiled resources.

- **Limitations**: Currently limited to models of *amount* depending on *structural-unit-size*
- **Dependencies**: *Complexity Collector*

Regression analyzer tries to find a fitting model to estimate the *amount* of resources depending on *structural-unit-size*.

The following strategies are currently available:

1. **Full Computation** uses all of the data points to obtain the best fitting model for each type of model from the database (unless `--regression_models`/`-r` restrict the set of models)
2. **Iterative Computation** uses a percentage of data points to obtain some preliminary models together with their errors or fitness. The most fitting model is then expanded, until it is fully computed or some other model becomes more fitting.
3. **Full Computation with initial estimate** first uses some percent of data to estimate which model would be best fitting. Given model is then fully computed.
4. **Interval Analysis** uses more finer set of intervals of data and estimates models for each interval providing more precise modeling of the profile.

5. **Bisection Analysis** fully computes the models for full interval. Then it does a split of the interval and computes new models for them. If the best fitting models changed for sub intervals, then we continue with the splitting.

Currently we support **linear**, **quadratic**, **power**, **logaritmic** and **constant** models and use the *coeficient of determination* (R^2) to measure the fitness of model. The models are stored as follows:

```
{
    "uid": "SLList_insert(SLList*, int)",
    "r_square": 0.0017560012128507133,
    "coeffs": [
        {
            "value": 0.505375215875552,
            "name": "b0"
        },
        {
            "value": 9.935159839322705e-06,
            "name": "b1"
        }
    ],
    "x_interval_start": 0,
    "x_interval_end": 11892,
    "model": "linear",
    "method": "full",
}
```

For more details about regression analysis refer to *Regression Analysis*. For more details how to collect suitable resources refer to *Complexity Collector*.

```
perun postprocessby regression_analysis [OPTIONS]
```

Options

-m, --method <method>
Will use the <method> to find the best fitting models for the given profile. [required]

-r, --regression_models <regression_models>
Restricts the list of regression models used by the specified <method> to fit the data. If omitted, all regression models will be used in the computation.

-s, --steps <steps>
Restricts the number of number of steps / data parts used by the iterative, interval and initial guess methods

3.5 Show Commands

3.5.1 perun show

Interprets the given profile using the selected visualization technique.

Looks up the given profile and interprets it using the selected visualization technique. Some of the techniques outputs either to terminal (using `ncurses`) or generates HTML files, which can be browseable in the web browser (using `bokeh` library). Refer to concrete techniques for concrete options and limitations.

The shown <profile> will be looked up in the following steps:

1. If <profile> is in form `i@i` (i.e, an *index tag*), then *ith* record registered in the minor version <hash> index will be shown.
2. If <profile> is in form `i@p` (i.e., an *pending tag*), then *ith* profile stored in `.perun/jobs` will be shown.
3. <profile> is looked-up within the minor version <hash> index for a match. In case the <profile> is registered there, it will be shown.
4. <profile> is looked-up within the `.perun/jobs` directory. In case there is a match, the found profile will be shown.
5. Otherwise, the directory is walked for any match. Each found match is asked for confirmation by user.

Example 1. The following command will show the first profile registered at index of `HEAD~1` commit. The resulting graph will contain bars representing sum of amounts per each subtype of resources and will be shown in the browser:

```
perun show -m HEAD~1 0@i bars sum --of 'amount' --per 'subtype' -v
```

Example 2. The following command will show the profile at the absolute path using in raw JSON format:

```
perun show ./echo-time-hello-2017-04-02-13-13-34-12.perf raw
```

For a thorough list and description of supported visualization techniques refer to *Supported Visualizations*.

```
perun show [OPTIONS] <profile> COMMAND [ARGS]...
```

Options

-m, --minor <minor>
Will check the index of different minor version <hash> during the profile lookup

Arguments

<profile>
Required argument

3.5.2 Show units

perun show alloclist

Alloclist is a specific query interface for profiles collected by *memory* collector.

- **Limitations**: *memory* profiles generated by *Memory Collector*.
- **Interpretation style**: textual
- **Visualization backend**: terminal

Memstat contains several predefined functions for aggregated basic information about allocations within the profile—list of allocations, tops, sums, etc.—and serves as a base for future extension. Currently the following modes are supported:

1. `most` lists trace records sorted by the frequency of allocations of memory they made.
2. `sum` lists trace records sorted by the overall amount of allocated memory they made.

3. `func` lists allocation records w.r.t. to the specified functions.
4. `all` lists all allocation records in specified time interval sorted by timestamp.
5. `top` lists trace records sorted by amount of allocated memory.

The example of output of *alloclist* is as follows:

```
#1 malloc: 100B at 0x31341343
  └ malloc  in  /dir/subdir/file.c:32
    └ main  in /dir/subdir/file.c:12

#2 calloc: 12B at 0x31411341
  └ malloc in /dir/subdir/file.c:1
```

Note: alloclist is an old prototype and is expected to be replaced by more sophisticated queries in near future.

```
perun show alloclist [OPTIONS] MODE
```

Options

-t, --limit-to <limit_to>
The displayed list of records will be limited to <int> records.

--from-time <from_time>
The list of records will start from the <timestamp> (in seconds).

--to-time <to_time>
The list of records will be displayed until the <timestamp> (in seconds).

--function <function>
Specifies the name of the function for the `func` mode.

-a, --all-occurrences
Includes all of the occurrences of the function (even the partial participationin the call traces) in the list of allocations.

Arguments

MODE
Required argument

perun show bars

Customizable interpretation of resources using the bar format.

- **Limitations**: *none*.
- **Interpretation style**: graphical
- **Visualization backend**: Bokeh

Bars graph shows the aggregation (e.g. sum, count, etc.) of resources of given types (or keys). Each bar shows <func> of resources from <of> key (e.g. sum of amounts, average of amounts, count of types, etc.) per each <per> key (e.g. per each snapshot, or per each type). Moreover, the graphs can either be (i) stacked, where the different values of <by> key are shown above each other, or (ii) grouped, where the different values of <by> key are

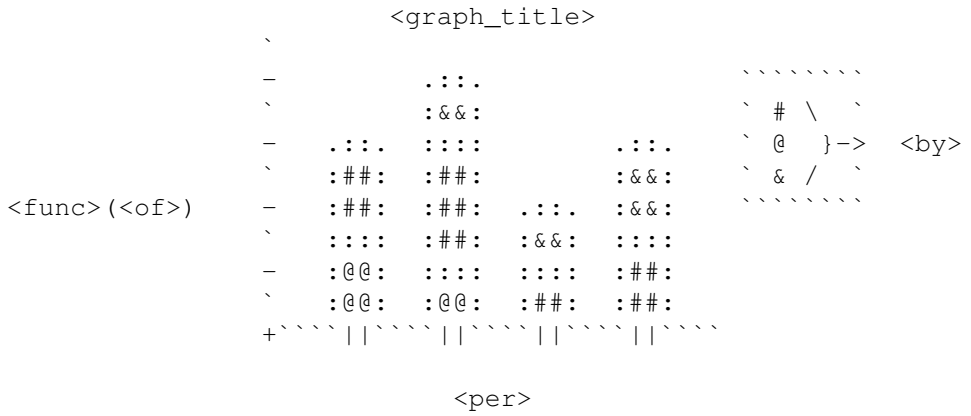
shown next to each other. Refer to `resources` for examples of keys that can be used as `<of>`, `<key>`, `<per>` or `<by>`.

Bokeh library is the current interpretation backend, which generates HTML files, that can be opened directly in the browser. Resulting graphs can be further customized by adding custom labels for axes, custom graph title or different graph width.

Example 1. The following will display the sum of sum of amounts of all resources of given for each subtype, stacked by uid (e.g. the locations in the program):

```
perun show 0@i bars sum --of 'amount' --per 'subtype' --stacked --by 'uid'
```

The example output of the bars is as follows:



Refer to *Bars Plot* for more thorough description and example of *bars* interpretation possibilities.

```
perun show bars [OPTIONS] <aggregation_function>
```

Options

-o, --of `<of_key>`
Sets key that is source of the data for the bars, i.e. what will be displayed on Y axis. [required]

-p, --per `<per_key>`
Sets key that is source of values displayed on X axis of the bar graph.

-b, --by `<by_key>`
Sets the key that will be used either for stacking or grouping of values

-s, --stacked
Will stack the values by <resource_key> specified by option –by.

-g, --grouped
Will stack the values by <resource_key> specified by option –by.

-f, --filename `<filename>`
Sets the outputs for the graph to the file.

-xl, --x-axis-label `<x_axis_label>`
Sets the custom label on the X axis of the bar graph.

-yl, --y-axis-label `<y_axis_label>`
Sets the custom label on the Y axis of the bar graph.

-gt, --graph-title <graph_title>
Sets the custom title of the bars graph.

-v, --view-in-browser
The generated graph will be immediately opened in the browser (firefox will be used).

Arguments

<aggregation_function>
Optional argument

perun show flamegraph

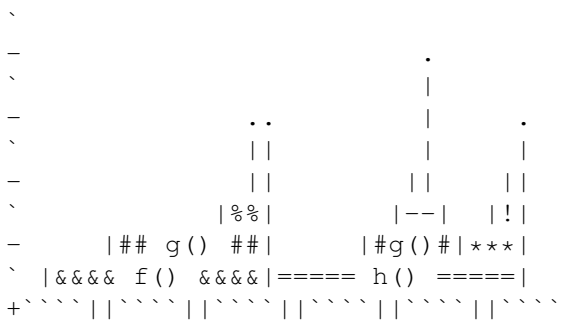
Flame graph interprets the relative and inclusive presence of the resources according to the stack depth of the origin of resources.

- **Limitations**: *memory* profiles generated by *Memory Collector*.
- **Interpretation style**: graphical
- **Visualization backend**: HTML

Flame graph intends to quickly identify hotspots, that are the source of the resource consumption complexity. On X axis, a relative consumption of the data is depicted, while on Y axis a stack depth is displayed. The wider the bars are on the X axis are, the more the function consumed resources relative to others.

Acknowledgements: Big thanks to Brendan Gregg for creating the original perl script for creating flame graphs w.r.t simple format. If you like this visualization technique, please check out this guy's site (http://brendangregg.com) for more information about performance, profiling and useful talks and visualization techniques!

The example output of the flamegraph is more or less as follows:



Refer to *Flame Graph* for more thorough description and examples of the interpretation technique. Refer to *perun.profile.convert.to_flame_graph_format()* for more details how the profiles are converted to the flame graph format.

```
perun show flamegraph [OPTIONS]
```

Options

-f, --filename <filename>
Sets the output file of the resulting flame graph.

-h, --graph-height <graph_height>
Increases the width of the resulting flame graph.

perun show flow

Customizable interpretation of resources using the flow format.

- **Limitations**: *none*.
- **Interpretation style**: graphical, textual
- **Visualization backend**: Bokeh, ncurses

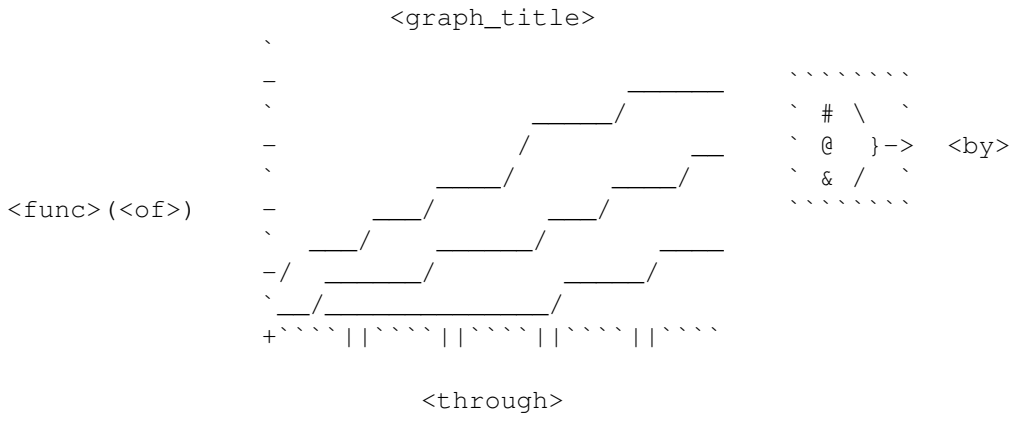
Flow graph shows the values resources depending on the independent variable as basic graph. For each group of resources identified by unique value of <by> key, one graph shows the dependency of <of> values aggregated by <func> depending on the <through> key. Moreover, the values can either be accumulated (this way when displaying the value of 'n' on x axis, we accumulate the sum of all values for all m < n) or stacked, where the graphs are output on each other and then one can see the overall trend through all the groups and proportions between each of the group.

Bokeh library is the current interpretation backend, which generates HTML files, that can be opened directly in the browser. Resulting graphs can be further customized by adding custom labels for axes, custom graph title or different graph width.

Example 1. The following will show the average amount (in this case the function running time) of each function depending on the size of the structure over which the given function operated:

```
perun show 0@i flow mean --of 'amount' --per 'structure-unit-size'
    --acumulated --by 'uid'
```

The example output of the bars is as follows:



Refer to *Flow Plot* for more thorough description and example of *flow* interpretation possibilities.

```
perun show flow [OPTIONS] <aggregation_function>
```

Options

-o, --of <of_key>
Sets key that is source of the data for the flow, i.e. what will be displayed on Y axis, e.g. the amount of resources. [required]

-t, --through <through_key>
Sets key that is source of the data value, i.e. the independent variable, like e.g. snapshots or size of the structure.

-b, --by <by_key>
For each <by_resource_key> one graph will be output, e.g. for each subtype or for each location of resource. [required]

-s, --stacked
Will stack the y axis values for different <by> keys on top of each other. Additionaly shows the sum of the values.

--accumulate, --no-accumulate
Will accumulate the values for all previous values of X axis.

-ut, --use-terminal
Shows flow graph in the terminal using ncurses library.

-f, --filename <filename>
Sets the outputs for the graph to the file.

-xl, --x-axis-label <x_axis_label>
Sets the custom label on the X axis of the flow graph.

-yl, --y-axis-label <y_axis_label>
Sets the custom label on the Y axis of the flow graph.

-gt, --graph-title <graph_title>
Sets the custom title of the flow graph.

-v, --view-in-browser
The generated graph will be immediately opened in the browser (firefox will be used).

Arguments

<aggregation_function>
Optional argument

perun show heapmap

Shows interactive map of memory allocations to concrete memories for each function.

- **Limitations**: *memory* profiles generated by *Memory Collector*.
- **Interpretation style**: textual
- **Visualization backend**: ncurses

Heap map shows the underlying memory map, and links the concrete allocations to allocated addresses for each snapshot. The map is interactive, one can either play the full animation of the allocations through snapshots or move and explore the details of the map.

Moreover, the heap map contains *heat map* mode, which accumulates the allocations into the heat representation—the hotter the colour displayed at given memory cell, the more time it was allocated there.

The heap map aims at showing the fragmentation of the memory and possible differences between different allocation strategies. On the other hand, the heat mode aims at showing the bottlenecks of allocations.

Refer to *Heap Map* for more thorough description and example of *heapmap* interpretation possibilities.

```
perun show heapmap [OPTIONS]
```

perun show scatter

Interactive visualization of resources and models in scatter plot format.

Scatter plot shows resources as points according to the given parameters. The plot interprets <per> and <of> as x, y coordinates for the points. The scatter plot also displays models located in the profile as a curves/lines.

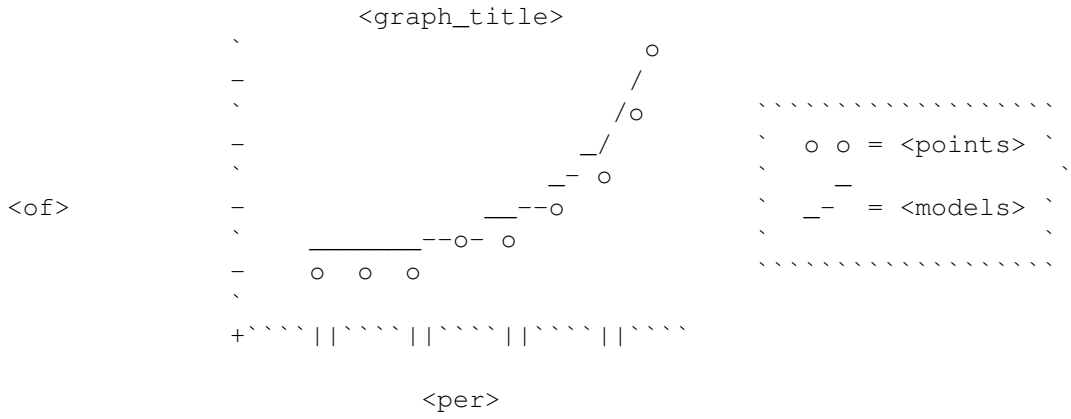
- **Limitations**: *none*.
- **Interpretation style**: graphical
- **Visualization backend**: Bokeh

Features in progress:

- uid filters
- models filters
- multiple graphs interpretation

Graphs are displayed using the Bokeh library and can be further customized by adding custom labels for axis, custom graph title and different graph width.

The example output of the scatter is as follows:



Refer to *Scatter Plot* for more thorough description and example of *scatter* interpretation possibilities. For more thorough explanation of regression analysis and models refer to *Regression Analysis*.

```
perun show scatter [OPTIONS]
```

Options

-o, --of <of_key>
Data source for the scatter plot, i.e. what will be displayed on Y axis. [default: amount]

-p, --per <per_key>
Keys that will be displayed on X axis of the scatter plot. [default: structure-unit-size]

-f, --filename <filename>
Outputs the graph to the file specified by filename.

-xl, --x-axis-label <x_axis_label>
Label on the X axis of the scatter plot.

-yl, --y-axis-label <y_axis_label>
Label on the Y axis of the scatter plot.

-gt, --graph-title <graph_title>
Title of the scatter plot.

-v, --view-in-browser
Will show the graph in browser.

3.6 Utility Commands

3.6.1 perun utils

Contains set of developer commands, wrappers over helper scripts and other functions that are not the part of the main perun suite.

```
perun utils [OPTIONS] COMMAND [ARGS]...
```

Commands

create
According to the given <template> constructs...

3.6.2 perun utils create

According to the given <template> constructs a new modules in Perun for <unit>.

Currently this supports creating new modules for the tool suite (namely `collect`, `postprocess`, `view`) or new algorithms for checking degradation (check). The command uses templates stored in *..perun emplates* directory and uses _jinja as a template handler. The templates can be parametrized by the following by options (if not specified 'none' is used).

Unless `--no-edit` is set, after the successful creation of the files, an external editor, which is specified by *general.editor* configuration key.

```
perun utils create [OPTIONS] <template> <unit>
```

Options

-nb, --no-before-phase
If set to true, the unit will not have before() function defined.

-na, --no-after-phase
If set to true, the unit will not have after() function defined.

--author <author>
Specifies the author of the unit

-ne, --no-edit
Will open the newly created files in the editor specified by *general.editor* configuration key.

-st, --supported-type <supported_types>
Sets the supported types of the unit (i.e. profile types).

Arguments

<template>
Required argument

<unit>
Required argument

CHAPTER

FOUR

COLLECTORS OVERVIEW

Performance profiles originate either from the user's own means (i.e. by building their own collectors and generating the profiles w.r.t *Specification of Profile Format*) or using one of the collectors from Perun's tool suite.

Perun can collect profiling data in two ways:

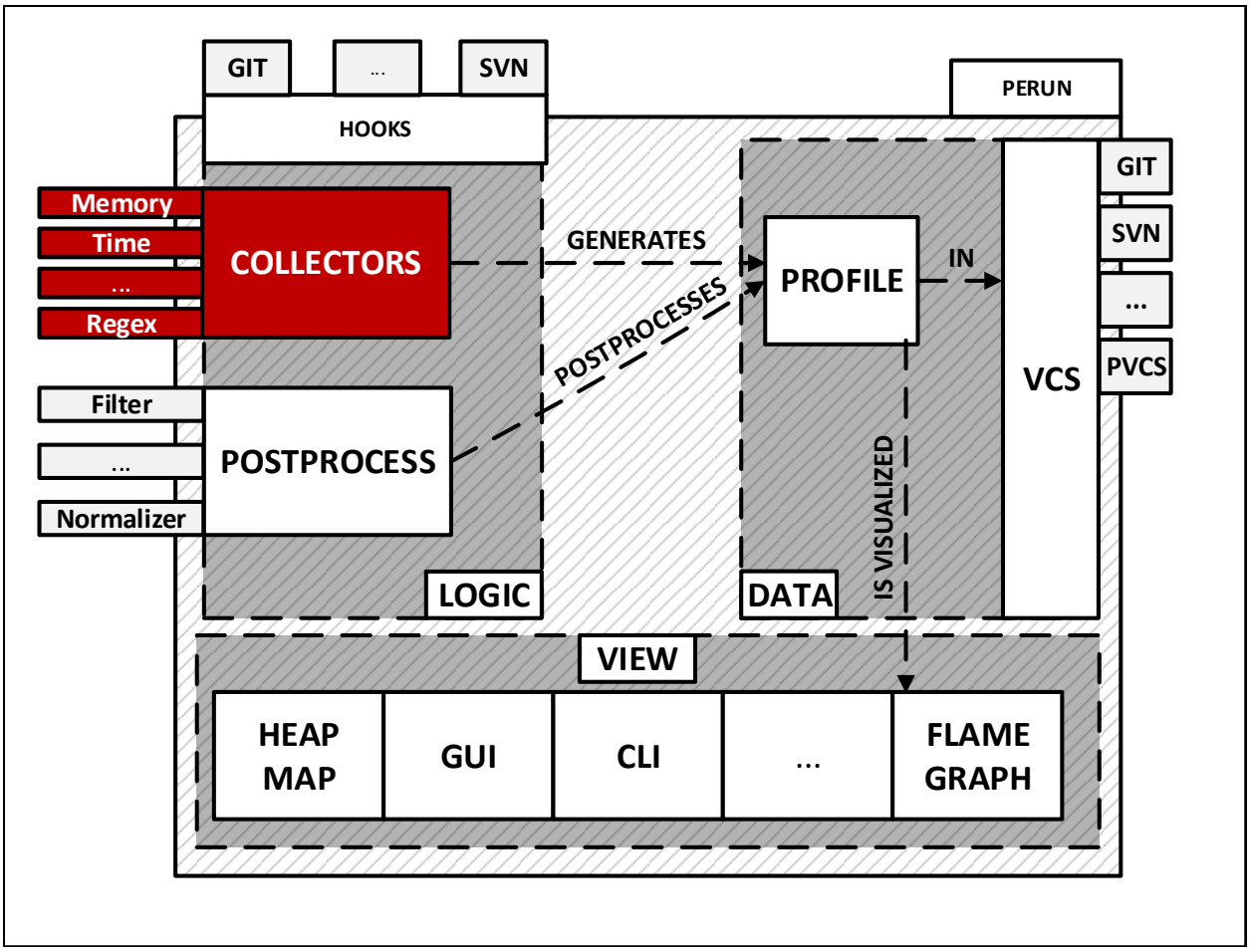
1. By **Directly running collectors** through `perun collect` command, that generates profile using a single collector with given collector configuration. The resulting profiles are not postprocessed in any way.
2. By **Using job specification** either as a single run of batch of profiling jobs using `perun run job` or according to the specification of the so called job matrix using `perun run matrix` command.

The format of resulting profiles is w.r.t. *Specification of Profile Format*. The *origin* is set to the current `HEAD` of the wrapped repository. However, note that uncommited changes may skew the resulting profile and Perun cannot guard your project against this. Further, *collector_info* is filled with configuration of the run collector.

All of the automatically generated profiles are stored in the `.perun/jobs/` directory as a file with the `.perf` extension. The filename is by default automatically generated according to the following template:

```
bin-collector-workload-timestamp.perf
```

Profiles can be further registered and stored in persistent storage using `perun add` command. Then both stored and pending profiles (i.e. those not yet assigned) can be postprocessed using the `perun postprocessby` or interpreted using available interpretation techniques using `perun show`. Refer to *Command Line Interface*, *Postprocessors Overview* and *Visualizations Overview* for more details about running command line commands, capabilities of postprocessors and interpretation techniques respectively. Internals of perun storage is described in *Perun Internals*.



4.1 Supported Collectors

Perun's tool suite currently contains the following three collectors:

1. *Complexity Collector* (authored by **Jirka Pavela**), collects running times of C/C++ functions along with the size of the structures they were executed on. E.g. this collects resources such that function `search` over the class `SingleLinkedList` took 100ms on single linked list with one million elements. *Examples* shows concrete examples of profiles generated by *Complexity Collector*
2. *Memory Collector* (authored by **Radima Podola**), collects specifications of allocations in C/C++ programs, such as the type of allocation or the full call trace. *Examples* shows concrete generated profiles by *Memory Collector*.
3. *Time Collector*, collects overall running times of arbitrary commands. Internally implemented as a simple wrapper over `time` utility

All of the listed collectors can be run from command line. For more information about command line interface for individual collectors refer to *Collect units*.

Collector modules are implementation independent (hence, can be written in any language) and only requires simple python interface registered within Perun. For brief tutorial how to create and register new collectors in Perun refer to *Creating your own Collector*.

4.1.1 Complexity Collector

Complexity collector collects running times of C/C++ functions along with the sizes of the structures they were executed on. The collected data are suitable for further postprocessing using the regression analysis and visualization by scatter plots.

Overview and Command Line Interface

perun collect complexity

Generates *complexity* performance profile, capturing running times of function depending on underlying structural sizes.

- **Limitations**: C/C++ binaries
- **Metric**: *mixed* (captures both *time* and *size* consumption)
- **Dependencies**: `SystemTap` (+ corresponding requirements e.g. kernel -dbgsym version)
- **Default units**: *us* for *time*, *element number* for *size*

Example of collected resources is as follows:

```
{
    "amount": 11,
    "subtype": "time delta",
    "type": "mixed",
    "uid": "SLList_init(SLList*)",
    "structure-unit-size": 0
}
```

Complexity collector provides various collection *strategies* which are supposed to provide sensible default settings for collection. This allows the user to choose suitable collection method without the need of detailed rules / sampling specification. Currently supported strategies are:

- **userspace**: This strategy traces all userspace functions / code blocks without

the use of sampling. Note that this strategy might be resource-intensive. * **all**: This strategy traces all userspace + library + kernel functions / code blocks that are present in the traced binary without the use of sampling. Note that this strategy might be very resource-intensive. * **u_sampled**: Sampled version of the **userspace** strategy. This method uses sampling to reduce the overhead and resources consumption. * **a_sampled**: Sampled version of the **all** strategy. Its goal is to reduce the overhead and resources consumption of the **all** method. * **custom**: User-specified strategy. Requires the user to specify rules and sampling manually.

Note that manually specified parameters have higher priority than strategy specification and it is thus possible to override concrete rules / sampling by the user.

Complexity profiles are suitable for postprocessing by *Regression Analysis* since they capture dependency of time consumption depending on the size of the structure. This allows one to model the estimation of complexity of individual functions.

Scatter plots are suitable visualization for profiles collected by *complexity* collector, which plots individual points along with regression models (if the profile was postprocessed by regression analysis). Run `perun show scatter --help` or refer to *Scatter Plot* for more information about *scatter plots*.

Refer to *Complexity Collector* for more thorough description and examples of *complexity* collector.

```
perun collect complexity [OPTIONS]
```

Options

-m, --method <method>
Select strategy for probing the binary. See documentation for detailed explanation for each strategy. [required]

-r, --rules <rules>
Set the probe points for profiling.

-s, --sampling <sampling>
Set the runtime sampling of the given probe points.

-g, --global_sampling <global_sampling>
Set the global sample for all probes, –sampling parameter for specific rules have higher priority.

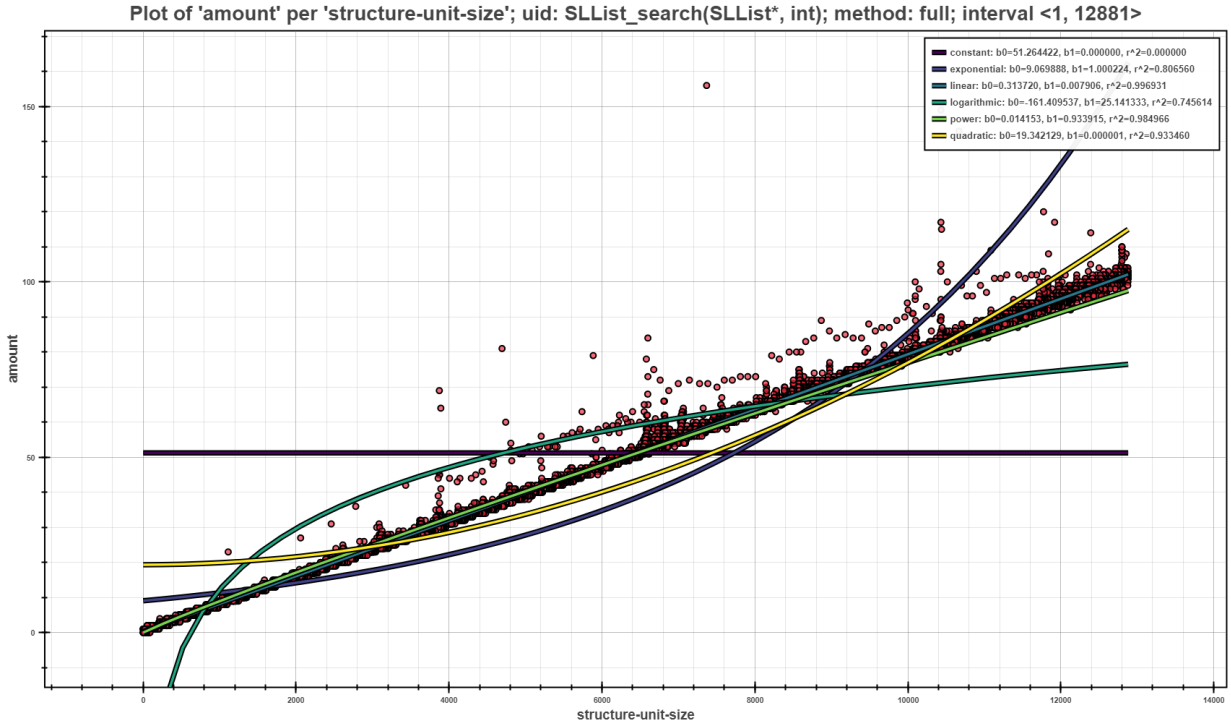
Examples


```
{
  "origin": "f7f3dcea69b97f2b03c421a223a770917149cfae",
  "header": {
    "cmd": "../stap-collector/tst",
    "type": "mixed",
    "units": {
      "mixed(time delta)": "us"
    },
    "workload": "",
    "params": ""
  },
  "collector_info": {
    "name": "complexity",
    "params": {
      "rules": [
        "SLList_init",
        "SLList_insert",
        "SLList_search",
        "SLList_destroy"
      ],
      "sampling": [
        {
          "func": "SLList_insert",
          "sample": 1
        },
        {
          "func": "func1",
          "sample": 1
        }
      ],
      "method": "custom",
      "global_sampling": null
    }
  },
  "postprocessors": [],
  "global": {
    "time": "6.8e-05s",
```

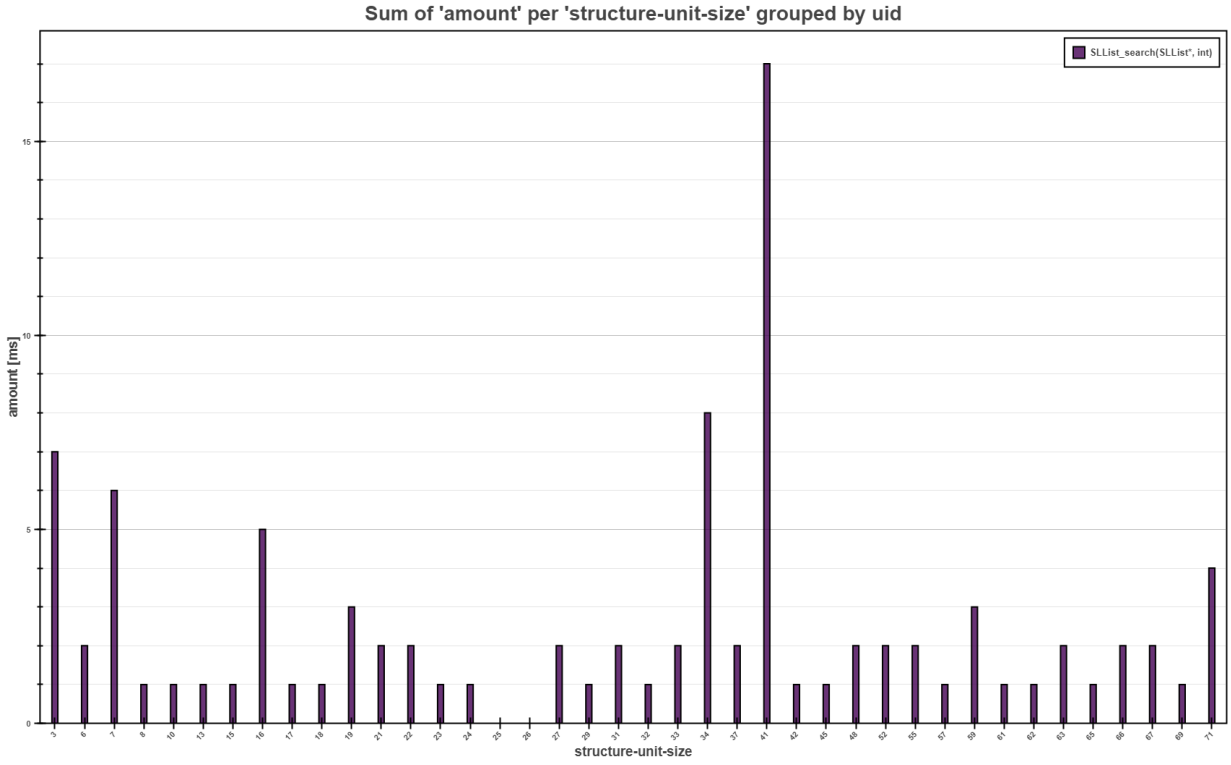
```
    "resources": [
      {
        "type": "mixed",
        "amount": 6,
        "subtype": "time delta",
        "uid": "SLList_init(SLList*)",
        "structure-unit-size": 0
      },
      {
        "type": "mixed",
        "amount": 0,
        "subtype": "time delta",
        "uid": "SLList_search(SLList*, int)",
        "structure-unit-size": 0
      },
      {
        "type": "mixed",
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 0
      },
      {
        "type": "mixed",
        "amount": 0,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 1
      },
      {
        "type": "mixed",
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 2
      },
      {
        "type": "mixed",
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 3
      },
      {
        "type": "mixed",
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_destroy(SLList*)",
        "structure-unit-size": 4
      }
    ]
  }
}
```


The above is an example of profiled data for the simple manipulation with program with single linked list. Profile captured running times of three functions—`SLList_init` (an initialization of single linked list), `SLList_destroy` (a destruction of single linked list) and `SLList_search` (search over the single linked list).

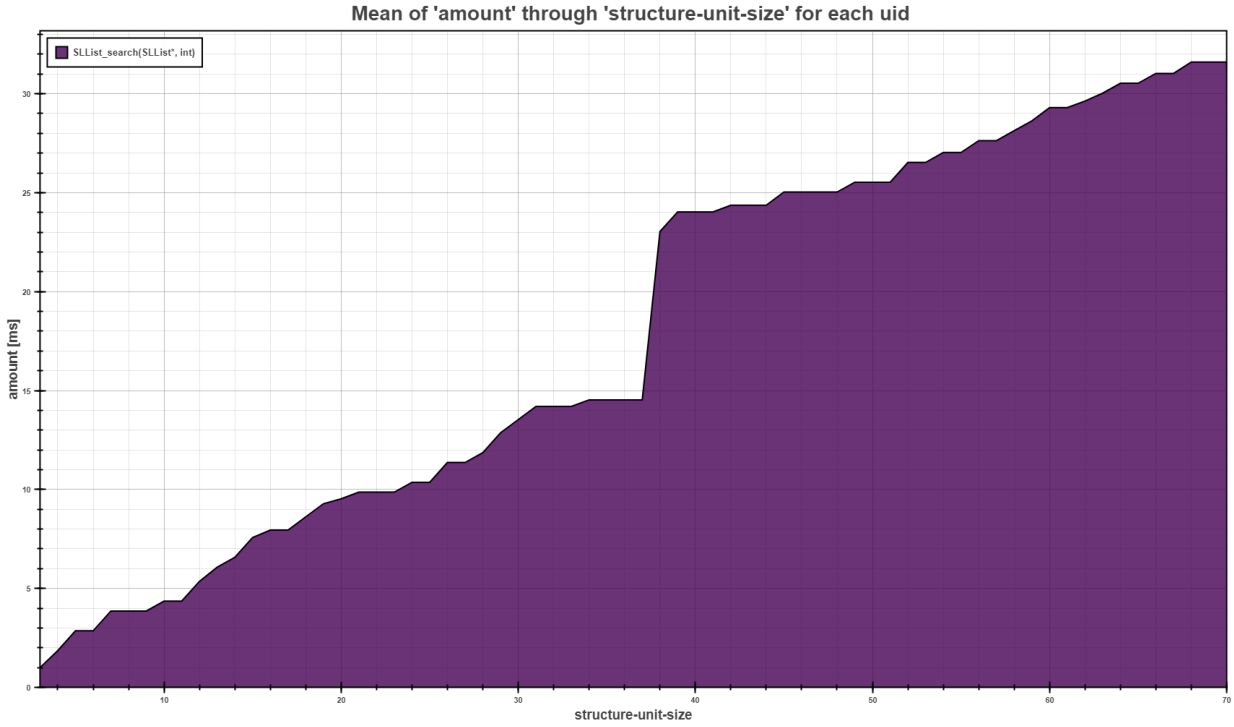
Highlighted lines show important keys and regions in the profile, e.g. the *origin*, collector-info or resources.



The *Scatter Plot* above shows the example of visualization of complexity profile. Each points corresponds to the running time of the SLList_search function over the single linked list with structure-unit-size elements. Elements are further interleaved with set of models obtained by *Regression Analysis*. The light green line corresponds to *linear* model, which seems to be the most fitting to model the performance of given function.



The *Bars Plot* above shows the overall sum of the running times for each `structure-unit-size` for the `SLList_search` function. The interpretation highlights that the most of the consumed running time were over the single linked lists with 41 elements.



The *Flow Plot* above shows the trend of the average running time of the `SLList_search` function depending on the size of the structure we execute the search on.

4.1.2 Memory Collector

Memory collector collects allocations of C/C++ functions, target addresses of allocations, type of allocations, etc. The collected data are suitable for visualiation using e.g. *Heap Map*.

Overview and Command Line Interface

perun collect memory

Generates *memory* performance profiel, capturing memory allocations of different types along with target address and full call trace.

- **Limitations**: C/C++ binaries
- **Metric**: *memory*
- **Dependencies**: `libunwind.so` and custom `libmalloc.so`
- **Default units**: *B* for *memory*

The following snippet shows the example of resources collected by *memory* profiler. It captures allocations done by functions with more detailed description, such as the type of allocation, trace, etc.

```
{
    "type": "memory",
    "subtype": "malloc",
    "address": 19284560,
    "amount": 4,
    "trace": [
        {
            "source": "../memory_collect_test.c",
            "function": "main",
            "line": 22
        },
    ],
    "uid": {
        "source": "../memory_collect_test.c",
        "function": "main",
        "line": 22
    }
},
```

Memory profiles can be efficiently interpreted using *Heap Map* technique (together with its *heat* mode), which shows memory allocations (by functions) in memory address map.

Refer to *Memory Collector* for more thorough description and examples of *memory* collector.

```
perun collect memory [OPTIONS]
```

Options

-s, --sampling <sampling>
: Sets the sampling interval for profiling the allocations. I.e. memory snapshots will be collected each <sampling> seconds.

--no-source <no_source>
: Will exclude allocations done from <no_source> file during the profiling.

--no-func <no_func>
: Will exclude allocations done by <no func> function during the profiling.

-a, --all
: Will record the full trace for each allocation, i.e. it will include all allocators and even unreachable records.

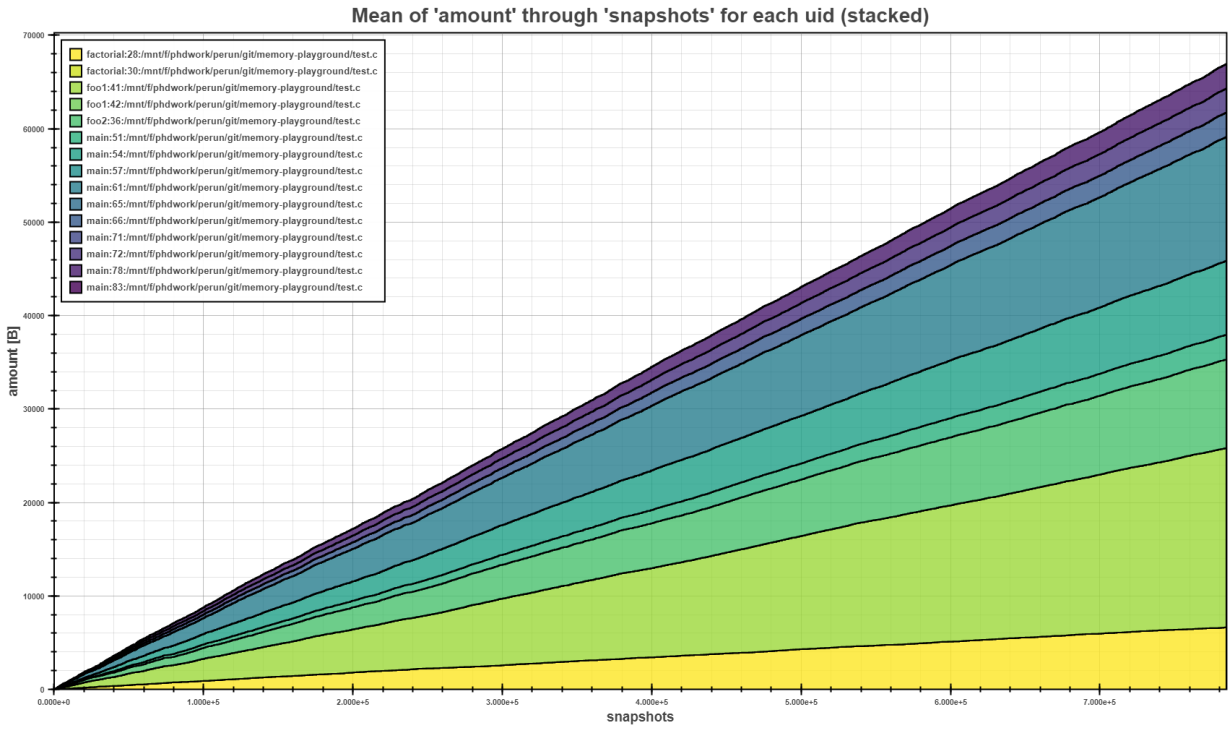
Examples


```
{
  "origin": "74288675e4074f1ad5bbb0d3b3253911ab42267a",
  "header": {
    "type": "memory",
    "workload": "",
    "cmd": "./mct",
    "units": {
      "memory": "B"
    },
    "params": ""
  },
  "collector_info": {
    "name": "memory",
    "params": {
      "no_func": null,
      "no_source": null,
      "all": false,
      "sampling": 0.025
    }
  },
  "postprocessors": [],
  "snapshots": [
    {
      "resources": [
      {
          "type": "memory",
          "subtype": "malloc",
          "address": 19284560,
          "amount": 4,
          "trace": [
            {
              "source": "unreachable",
              "function": "malloc",
              "line": 0
            },
            {
              "source": "../memory_collect_test.c",
              "function": "main",
              "line": 22
            },
```

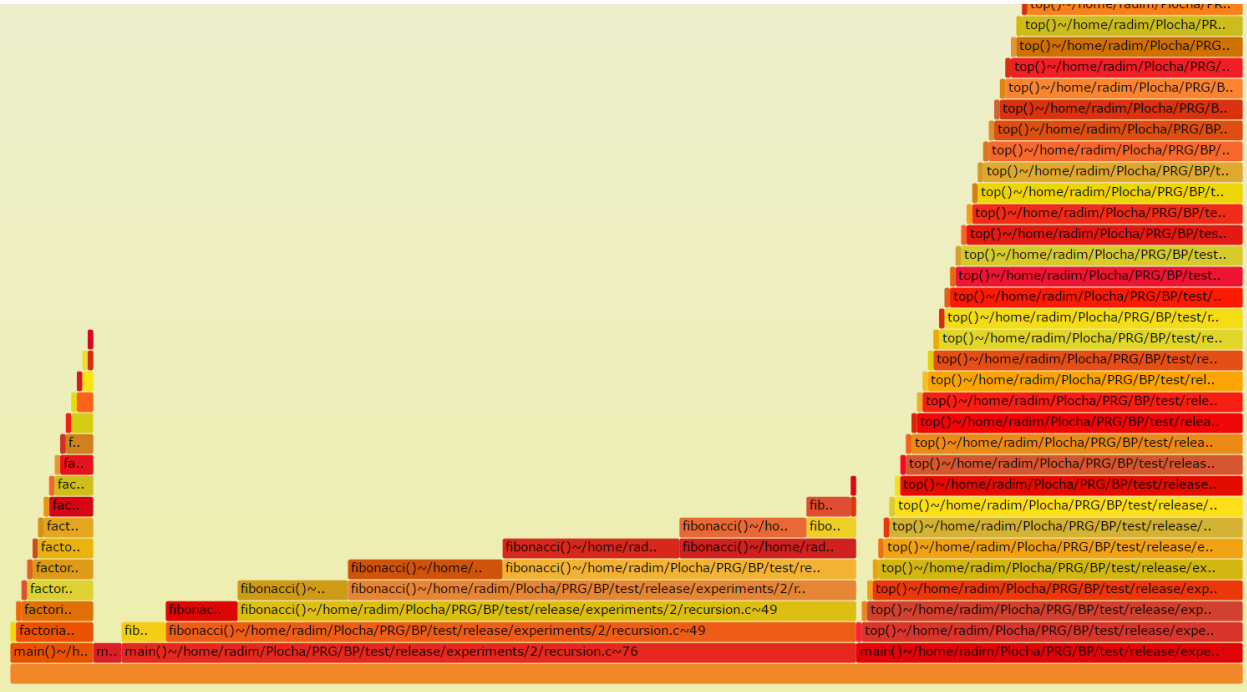
```
            {
              "source": "unreachable",
              "function": "__libc_start_main",
              "line": 0
            },
            {
              "source": "unreachable",
              "function": "_start",
              "line": 0
            }
          ],
          "uid": {
            "source": "../memory_collect_test.c",
            "function": "main",
            "line": 22
          }
        },
        {
          "type": "memory",
          "subtype": "free",
          "address": 19284560,
          "amount": 0,
          "trace": [
            {
              "source": "unreachable",
              "function": "free",
              "line": 0
            },
            {
              "source": "../memory_collect_test.c",
              "function": "main",
              "line": 27
            },
            {
              "source": "unreachable",
              "function": "__libc_start_main",
              "line": 0
            },
            {
              "source": "unreachable",
              "function": "_start",
              "line": 0
            }
          ],
          "uid": {
            "source": "../memory_collect_test.c",
            "function": "main",
            "line": 27
          }
        }
      ],
      "time": "0.025000"
    }
  ]
}
```


The above is an example of profiled data on a simple binary, which makes several minor allocations. Profile shows a simple allocation followed by deallocation and highlights important keys and regions in the *memory* profiles, e.g. the

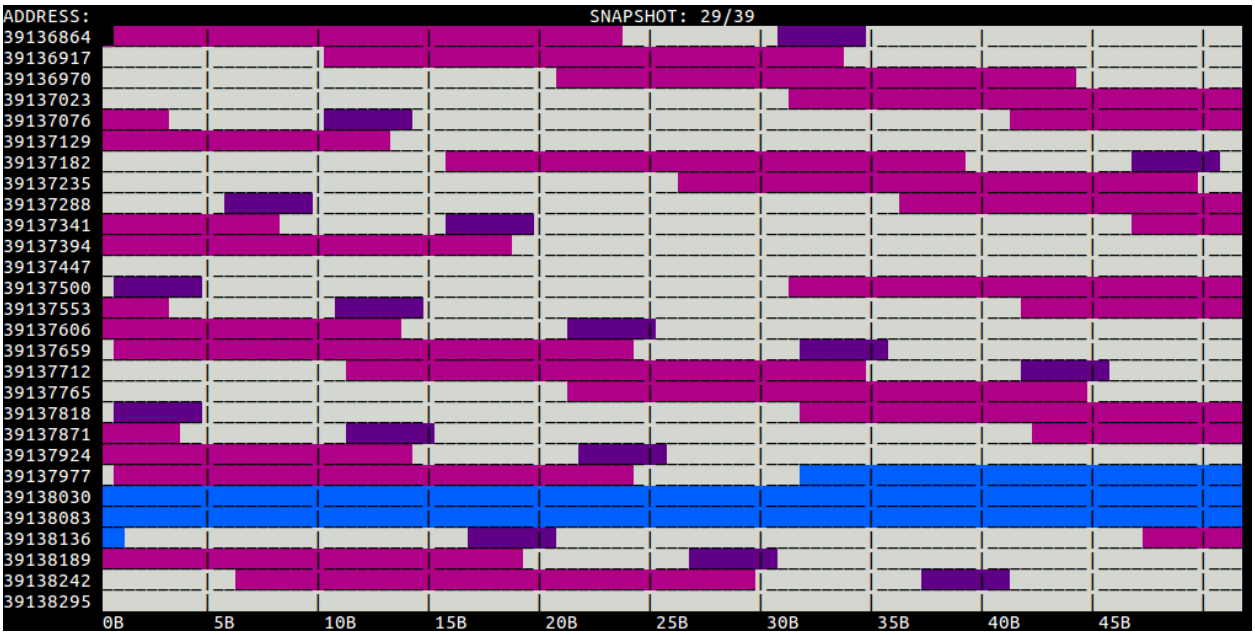
origin, collector-info or resources



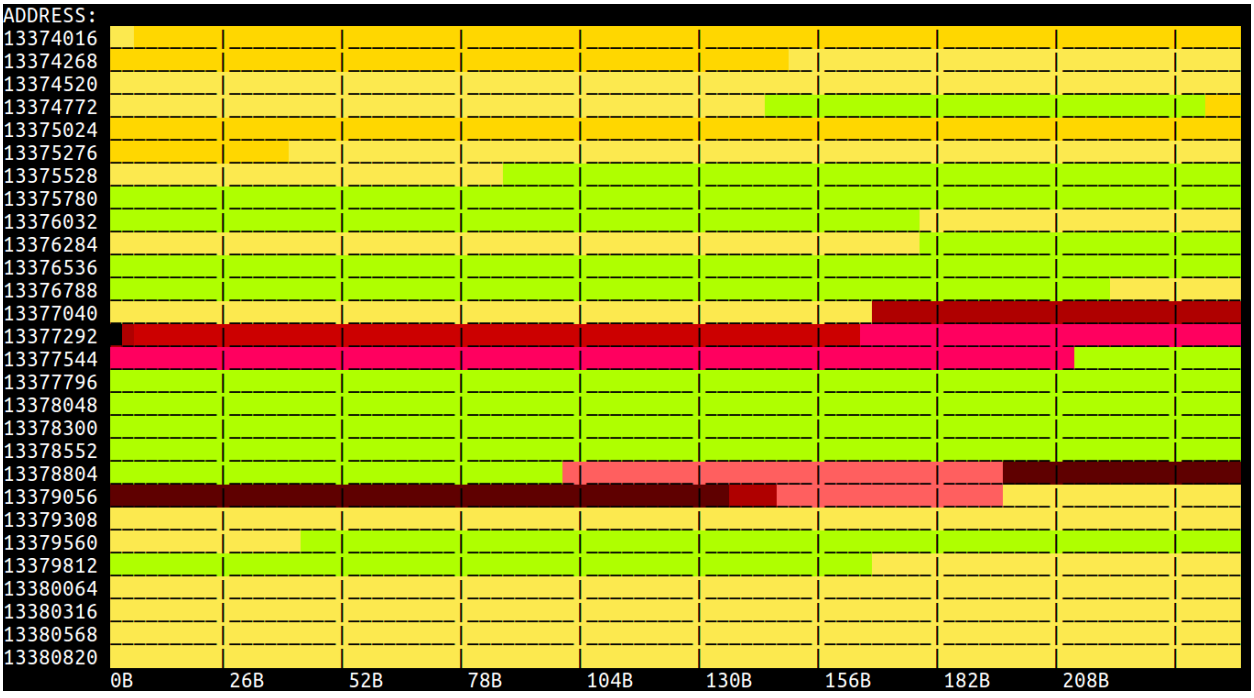
The *Flow Plot* above shows the mean of allocated amounts per each allocation site (i.e. uid) in stacked mode. The stacking of the means clearly shows, where the biggest allocations where made during the program run.



The *Flame Graph* is an efficient visualization of inclusive consumption of resources. The width of the base of one flame shows the bottleneck and hotspots of profiled binaries.



The *Heap Map* shows the address space through the time (snapshots) and visualize the fragmentation of memory allocation per each allocation site. The *heap map* aboe shows the difference between allocations using lists (purple), skiplists (pinkish) and standard vectors (blue). The map itself is interactive and displays details about individual address cells.



Heat map is a mode of heap map, which aggregates the allocations over all of the snapshots and uses warmer colours for address cells, where more allocations were performed.

4.1.3 Time Collector

Time collector collects is a simple wrapper over the time utility. There is nothing special about this, the profiles are simple, and no visualization is especially suitable for this mode.

Overview and Command Line Interface

perun collect time

Generates *time* performance profile, capturing overall running times of the profiled command.

- **Limitations**: *none*
- **Metric**: running *time*
- **Dependencies**: *none*
- **Default units**: *s*

This is a wrapper over the `time` linux unitility and captures resources in the following form:

```
{
    "amount": 0.59,
    "type": "time",
    "uid": "sys"
}
```

Refer to *Time Collector* for more thorough description and examples of *complexity* collector.

```
perun collect time [OPTIONS]
```

Examples

```
{
  "origin": "9524dd21a51ed5d462b802fd3bc7e070462cceac",
  "header": {
    "params": "status",
    "type": "time",
    "cmd": "perun",
    "workload": "--short",
    "units": {
      "time": "s"
    }
  },
  "collector_info": {
    "params": {},
    "name": "time"
  },
  "postprocessors": [],
  "global": {
    "timestamp": "2.32",
    "resources": [
      {
        "amount": 0.59,
        "uid": "sys"
```

```
      },
      {
        "amount": 0.32,
        "uid": "user"
      },
      {
        "amount": 2.32,
        "uid": "real"
      }
    ]
  },
}
```

The above is an example of profiled data using the *time* wrapper with important regions and keys highlighted.

4.2 Creating your own Collector

New collectors can be registered within Perun in several steps. Internally they can be implemented in any programming language and in order to work with Perun requires three phases to be specified as given in *Collectors Overview*—`before()`, `collect()` and `after()`. Each new collector requires a interface module `run.py`, which contains the three functions and, moreover, a cli API for Click.

You can register your new collector as follows:

1. Run `perun utils create collect mycollector` to generate a new modules in `perun/collect` directory with the following structure. The command takes a predefined templates for new collectors and creates `__init__.py` and `run.py` according to the supplied command line arguments (see *Utility Commands* for more information about interface of `perun utils create` command):

```
/perun
|-- /collect
    |-- /mycollector
        |-- __init__.py
        |-- run.py
    |-- /complexity
    |-- /memory
    |-- /time
    |-- __init__.py
```

2. First, implement the `__init__.py` file, including the module docstring with brief collector descriptions and definitions of constants that are used for automatic setting of profiles (namely the `collector-info`) which has the following structure:

```
"""..."""

COLLECTOR_TYPE = 'time|memory|mixed'
COLLECTOR_DEFAULT_UNITS = {
    'type': 'unit'
}

__author__ = 'You!'
```

3. Next, implement the `run.py` module with `collect()` function, (optionally with `before()` and `after()`). The `collect()` function should do the actual collection of the profiling data over the given configuration. Each function should return the integer status of the phase, the status message

(used in case of error) and dictionary including params passed to additional phases and 'profile' with dictionary w.r.t *Specification of Profile Format*.

```
def before(**kwargs):
    """(optional)"""
    return STATUS, STATUS_MSG, dict(kwargs)


def collect(**kwargs):
    """..."""
    return STATUS, STATUS_MSG, dict(kwargs)


def after(**kwargs):
    """(optional)"""
    return STATUS, STATUS_MSG, dict(kwargs)
```

4. Additionally implement the command line interface function in `run.py`, named the same as your collector. This function will is called from command line as `perun collect mycollector` and is based on Click library.

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/collectors_run.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/collectors_run_api.py
@@ -1,3 +1,8 @@
+import click
+
+import perun.logic.runner as runner
+
+
 def before(**kwargs):
     """(optional)"""
     return STATUS, STATUS_MSG, dict(kwargs)
@@ -11,3 +16,10 @@
 def after(**kwargs):
     """(optional)"""
     return STATUS, STATUS_MSG, dict(kwargs)
+
+
+@click.command()
+@click.pass_context
+def mycollector(ctx, **kwargs):
+    """..."""
+    runner.run_collector_from_cli_context(ctx, 'mycollector', kwargs)
```

5. Finally register your newly created module in `get_supported_module_names()` located in `perun.utils.__init__.py`:

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪collectors.py
@@ -6,7 +6,7 @@
         ))
     return {
         'vcs': ['git'],
-        'collect': ['complexity', 'memory', 'time'],
+        'collect': ['complexity', 'memory', 'time', 'mycollector'],
         'postprocess': ['filter', 'normalizer', 'regression_analysis'],
         'view': ['alloclist', 'bars', 'flamegraph', 'flow', 'heapmap', 'raw
↪', 'scatter']
```

```
11      }[package]
```

6. Preferably, verify that registering did not break anything in the Perun and if you are not using the developer installation, then reinstall Perun:

```
make test
make install
```

7. At this point you can start using your collector either using `perun collect` or using the following to set the job matrix and run the batch collection of profiles:

```
perun config --edit
perun run matrix
```

8. If you think your collector could help others, please, consider making Pull Request.

CHAPTER

FIVE

POSTPROCESSORS OVERVIEW

Performance profiles originate either from the user's own means (i.e. by building their own collectors and generating the profiles w.r.t *Specification of Profile Format*) or using one of the collectors from Perun's tool suite.

Perun can postprocess such profiling data in two ways:

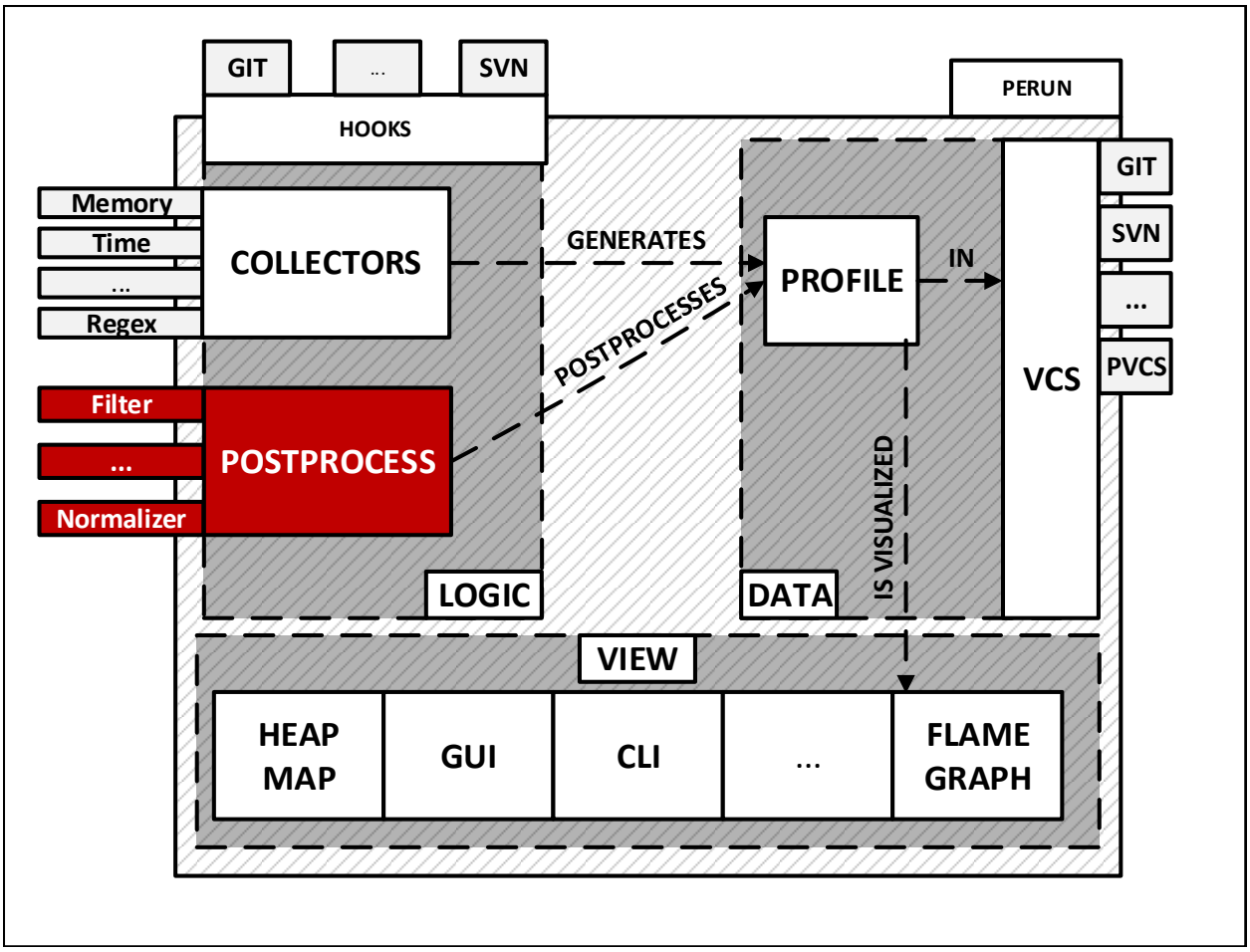
1. By **Directly running postprocessors** through `perun postprocessby` command, that takes the profile (either stored or pending) and uses a single postprocessor with given configuration.
2. By **Using job specification** either as a single run of batch of profiling jobs using `perun run job` or according to the specification of the so called job matrix using `perun run matrix` command.

The format of input and resulting profiles has to be w.r.t. *Specification of Profile Format*. By default new profiles are created. The *origin* set to the origin of the original profile. Further, *postprocessors* is extended with configuration of the run postprocessor (appended at the end).

All of the postprocessed profiles are stored in the `.perun/jobs/` directory as a file with the `.perf` extension. The filename is by default automatically generated according to the following template:

```
bin-collector-workload-timestamp.perf
```

Profiles can be further registered and stored in persistent storage using `perun add` command. Then both stored and pending profiles (i.e. those not yet assigned) can be interpreted using available interpretation techniques using `perun show`. Refer to *Command Line Interface* and *Visualizations Overview* for more details about running command line commands and capabilities fo interpretation techniques respectively. Internals of perun storage is described in *Perun Internals*.



5.1 Supported Postprocessors

Perun's tool suite currently contains the following two postprocessors:

1. *Normalizer Postprocessor* scales the resources of the given profile to the interval (0, 1). The main intuition behind the usage of this postprocessor is to be able to compare profiles from different workloads or parameters, which may have different scales of resource amounts.
2. *Regression Analysis* (authored by **Jirka Pavela**) attempts to do a regression analysis by finding the fitting model for dependent variable based on other independent one. Currently the postprocessor focuses on finding a well suited model (linear, quadratic, logarithmic, etc.) for the amount of time duration depending on size of the data structure the function operates on.

All of the listed postprocessors can be run from command line. For more information about command line interface for individual postprocessors refer to *Postprocess units*.

Postprocessors modules are implementation independent and only requires a simple python interface registered within Perun. For brief tutorial how to create and register your own postprocessors refer to *Examples*.

5.1.1 Normalizer Postprocessor

Command Line Interface

perun postprocessby normalizer

Normalizes performance profile into flat interval.

- **Limitations**: *none*
- **Dependencies**: *none*

Normalizer is a postprocessor, which iterates through all of the snapshots and normalizes the resources of same type to interval `(0, 1)`, where `1` corresponds to the maximal value of the given type.

Consider the following list of resources for one snapshot generated by *Time Collector*:

```
[
    {
        'amount': 0.59,
        'uid': 'sys'
    }, {
        'amount': 0.32,
        'uid': 'user'
    }, {
        'amount': 2.32,
        'uid': 'real'
    }
]
```

Normalizer yields the following set of resources:

```
[
    {
        'amount': 0.2543103448275862,
        'uid': 'sys'
    }, {
        'amount': 0.13793103448275865,
        'uid': 'user'
    }, {
        'amount': 1.0,
        'uid': 'real'
    }
]
```

Refer to *Normalizer Postprocessor* for more thorough description and examples of *normalizer* postprocessor.

```
perun postprocessby normalizer [OPTIONS]
```

5.1.2 Regression Analysis

Postprocessing of input profiles using the regression analysis. The regression analysis offers several computational methods and models for finding fitting models for trends in the captured profiling resources.

Command Line Interface

perun postprocessby regression_analysis

Finds fitting regression models to estimate models of profiled resources.

- **Limitations**: Currently limited to models of *amount* depending on *structural-unit-size*
- **Dependencies**: *Complexity Collector*

Regression analyzer tries to find a fitting model to estimate the *amount* of resources depending on *structural-unit-size*.

The following strategies are currently available:

1. **Full Computation** uses all of the data points to obtain the best fitting model for each type of model from the database (unless `--regression_models`/`-r` restrict the set of models)
2. **Iterative Computation** uses a percentage of data points to obtain some preliminary models together with their errors or fitness. The most fitting model is then expanded, until it is fully computed or some other model becomes more fitting.
3. **Full Computation with initial estimate** first uses some percent of data to estimate which model would be best fitting. Given model is then fully computed.
4. **Interval Analysis** uses more finer set of intervals of data and estimates models for each interval providing more precise modeling of the profile.
5. **Bisection Analysis** fully computes the models for full interval. Then it does a split of the interval and computes new models for them. If the best fitting models changed for sub intervals, then we continue with the splitting.

Currently we support **linear**, **quadratic**, **power**, **logaritmic** and **constant** models and use the *coeficient of determination* (R^2) to measure the fitness of model. The models are stored as follows:

```
{
    "uid": "SLList_insert(SLList*, int)",
    "r_square": 0.0017560012128507133,
    "coeffs": [
        {
            "value": 0.505375215875552,
            "name": "b0"
        },
        {
            "value": 9.935159839322705e-06,
            "name": "b1"
        }
    ],
    "x_interval_start": 0,
    "x_interval_end": 11892,
    "model": "linear",
    "method": "full",
}
```

For more details about regression analysis refer to *Regression Analysis*. For more details how to collect suitable resources refer to *Complexity Collector*.

```
perun postprocessby regression_analysis [OPTIONS]
```

Options

-m, --method <method>
Will use the <method> to find the best fitting models for the given profile. [required]

-r, --regression_models <regression_models>
Restricts the list of regression models used by the specified <method> to fit the data. If omitted, all regression models will be used in the computation.

-s, --steps <steps>
Restricts the number of number of steps / data parts used by the iterative, interval and initial guess methods

Examples

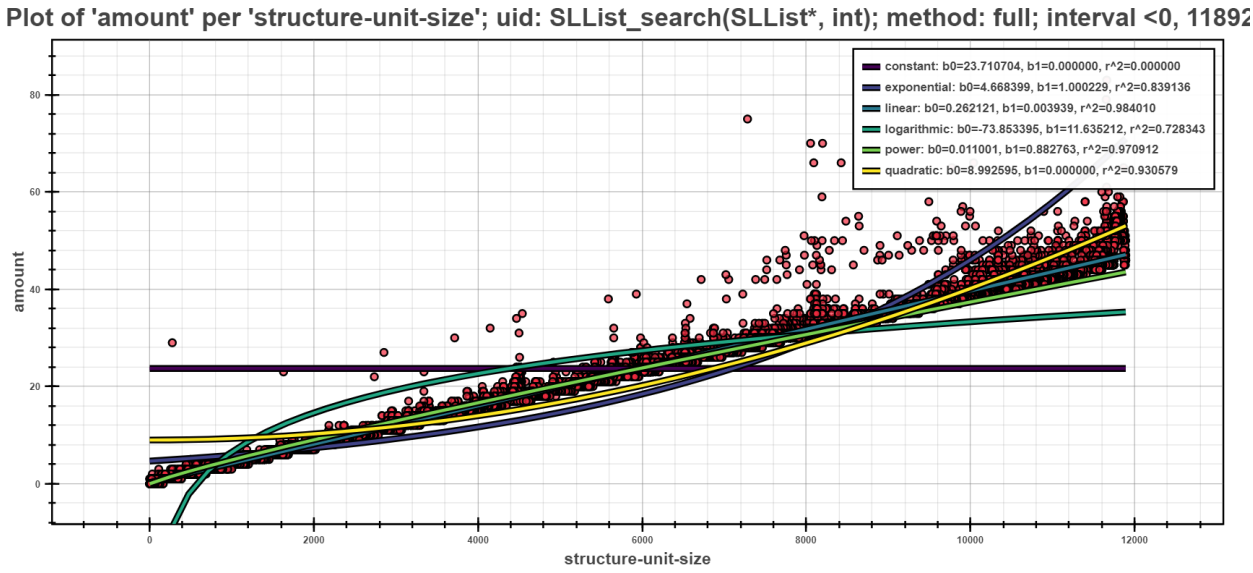

```
{
  "origin": "f7f3dcea69b97f2b03c421a223a770917149cfae",
  "header": {
    "cmd": "../stap-collector/tst",
    "type": "mixed",
    "units": {
      "mixed(time delta)": "us"
    },
    "workload": "",
    "params": ""
  },
  "collector_info": {
    "name": "complexity",
    "params": {
      "rules": [
        "SLList_init",
        "SLList_insert",
        "SLList_search",
        "SLList_destroy"
      ],
      "sampling": [
        {
          "func": "SLList_insert",
          "sample": 1
        },
        {
          "func": "func1",
          "sample": 1
        }
      ],
      "method": "custom",
      "global_sampling": null
    }
  },
  "postprocessors": [],
  "global": {
    "models": [
      {
        "coeffs": [
          {
            "value": 0.75,
            "name": "b0"
          },
```

```
            {
              "value": 0.0,
              "name": "b1"
            }
          ],
          "uid": "SLList_insert(SLList*, int)",
          "x_interval_end": 3,
          "model": "constant",
          "r_square": 0.0,
          "method": "full",
          "x_interval_start": 0
        },
        {
          "coeffs": [
            {
              "value": 1.0,
              "name": "b0"
            },
            {
              "value": 1.0,
              "name": "b1"
            }
          ],
          "uid": "SLList_insert(SLList*, int)",
          "x_interval_end": 3,
          "model": "exponential",
          "r_square": 0.0,
          "method": "full",
          "x_interval_start": 0
        },
        {
          "coeffs": [
            {
              "value": 0.6,
              "name": "b0"
            },
            {
              "value": 0.1,
              "name": "b1"
            }
          ],
          "uid": "SLList_insert(SLList*, int)",
          "x_interval_end": 3,
          "model": "linear",
          "r_square": 0.06666666666666667,
          "method": "full",
          "x_interval_start": 0
        },
        {
          "coeffs": [
            {
              "value": 0.08877935258260898,
              "name": "b0"
            },
            {
              "value": 0.9675751528184126,
              "name": "b1"
            }
```

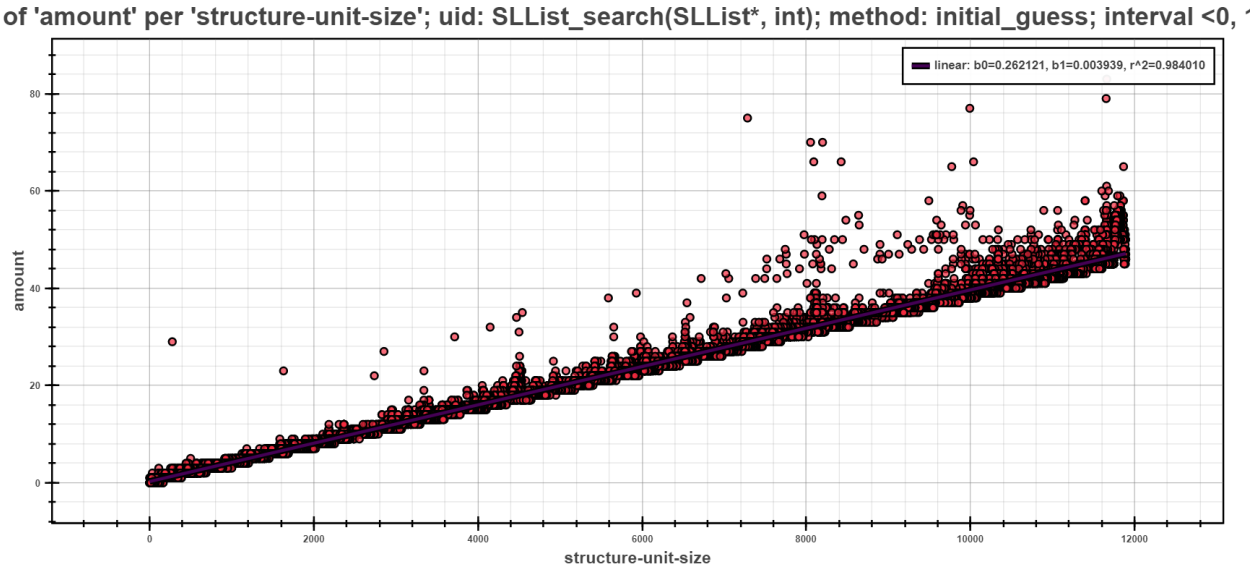
```
        ],
        "uid": "SLList_insert(SLList*, int)",
        "x_interval_end": 3,
        "model": "logarithmic",
        "r_square": 0.8668309711260865,
        "method": "full",
        "x_interval_start": 0
      },
      {
        "coeffs": [
          {
            "value": 1.0,
            "name": "b0"
          },
          {
            "value": 0.0,
            "name": "b1"
          }
        ],
        "uid": "SLList_insert(SLList*, int)",
        "x_interval_end": 3,
        "model": "power",
        "r_square": 0.0,
        "method": "full",
        "x_interval_start": 0
      },
      {
        "coeffs": [
          {
            "value": 0.5714285714285714,
            "name": "b0"
          },
          {
            "value": 0.05102040816326531,
            "name": "b1"
          }
        ],
        "uid": "SLList_insert(SLList*, int)",
        "x_interval_end": 3,
        "model": "quadratic",
        "r_square": 0.17006802721088435,
        "method": "full",
        "x_interval_start": 0
      }
    ],
    "resources": [
      {
        "amount": 6,
        "subtype": "time delta",
        "uid": "SLList_init(SLList*)",
        "structure-unit-size": 0,
        "type": "mixed"
      },
      {
        "amount": 0,
        "subtype": "time delta",
        "uid": "SLList_search(SLList*, int)",
        "structure-unit-size": 0,
```

```
        "type": "mixed"
      },
      {
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 0,
        "type": "mixed"
      },
      {
        "amount": 0,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 1,
        "type": "mixed"
      },
      {
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 2,
        "type": "mixed"
      },
      {
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_insert(SLList*, int)",
        "structure-unit-size": 3,
        "type": "mixed"
      },
      {
        "amount": 1,
        "subtype": "time delta",
        "uid": "SLList_destroy(SLList*)",
        "structure-unit-size": 4,
        "type": "mixed"
      }
    ],
    "time": "6.8e-05s"
  }
}
```


The profile above shows the complexity profile taken from *Examples* and postprocessed using the full method. The highlighted part shows all of the fully computed models of form $y = b_0 + b_1 * f(x)$, represented by their types (e.g. *linear*, *quadratic*, etc.), concrete found coeficients b_0 and b_1 and e.g. coeficient of determination R^2 for measuring the fitting of the model.



The *Scatter Plot* above shows the interpreted models of different complexity example, computed using the **full computation** method. In the picture, one can see that the depedency of running time based on the structural size is best fitted by *linear* models.



The next *scatter plot* displays the same data as previous, but regressed using the *initial guess* strategy. This strategy first does a computation of all models on small sample of data points. Such computation yields initial estimate of fitness of models (the initial sample is selected by random). The best fitted model is then chosen and fully computed on the rest of the data points.

The picture shows only one model, namely *linear* which was fully computed to best fit the given data points. The rest of the models had worse estimation and hence was not computed at all.

5.2 Creating your own Postprocessor

New postprocessors can be registered within Perun in several steps. Internally they can be implemented in any programming language and in order to work with perun requires one to three phases to be specified as given in *Postprocessors Overview*—`before()`, `postprocess()` and `after()`. Each new postprocessor requires a interface module `run.py`, which contains the three function and, moreover, a CLI function for Click framework.

You can register your new postprocessor as follows:

1. Run `perun utils create postprocess mypostprocessor` to generate a new modules in `perun/postprocess` directory with the following structure. The command takes a predefined templates for new postprocessors and creates `__init__.py` and `run.py` according to the supplied command line arguments (see *Utility Commands* for more information about interface of `perun utils create` command):

```
/perun
|-- /postprocess
    |-- /mypostprocessor
        |-- __init__.py
        |-- run.py
    |-- /normalizer
    |-- /regression_analysis
    |-- __init__.py
```

2. First, implement the `__init__py` file, including the module docstring with brief postprocessor description and definitions of constants that are used for internal checks which has the following structure:

```
"""..."""

SUPPORTED_PROFILES = ['mixed|memory|time']

__author__ = 'You!'
```

3. Next, implement the `run.py` module with `postprocess()` fucntion, (and optionally with `before()` and `after()` functions). The `postprocess()` function should do the actual postprocessing of the profile. Each function should return the integer status of the phase, the status message (used in case of error) and dictionary including params passed to additional phases and 'profile' with dictionary w.r.t. *Specification of Profile Format*.

```
def before(**kwargs):
    """(optional)"""
    return STATUS, STATUS_MSG, dict(kwargs)


def postprocess(profile, **configuration):
    """..."""
    return STATUS, STATUS_MSG, dict(kwargs)


def after(**kwargs):
    """(optional)"""
    return STATUS, STATUS_MSG, dict(kwargs)
```

4. Additionally, implement the command line interface function in `run.py`, named the same as your collector. This function will be called from the command line as `perun postprocessby mypostprocessor` and is based on Click_libary.

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/postprocess_run.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/postprocess_run_api.py
@@ -1,3 +1,8 @@
+import click
+
+import perun.logic.runner as runner
+
+
 def before(**kwargs):
     """(optional)"""
     return STATUS, STATUS_MSG, dict(kwargs)
@@ -11,3 +16,10 @@
 def after(**kwargs):
     """(optional)"""
     return STATUS, STATUS_MSG, dict(kwargs)
+
+
+@click.command()
+@pass_profile
+def regression_analysis(profile, **kwargs):
+    """..."""
+    runner.run_postprocessor_on_profile(profile, 'mypostprocessor', kwargs)
```

5. Finally register your newly created module in `get_supported_module_names()` located in `perun.utils.__init__.py`:

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪postprocess.py
@@ -7,6 +7,6 @@
     return {
         'vcs': ['git'],
         'collect': ['complexity', 'memory', 'time'],
-        'postprocess': ['filter', 'normalizer', 'regression_analysis'],
+        'postprocess': ['filter', 'normalizer', 'regression_analysis',
↪'mypostprocessor'],
         'view': ['alloclist', 'bars', 'flamegraph', 'flow', 'heapmap', 'raw
↪', 'scatter']
     }[package]
```

6. Preferably, verify that registering did not break anything in the Perun and if you are not using the developer installation, then reinstall Perun:

```
make test
make install
```

7. At this point you can start using your postprocessor either using `perun postprocessby` or using the following to set the job matrix and run the batch collection of profiles:

```
perun config --edit
perun run matrix
```

8. If you think your postprocessor could help others, please, consider making Pull Request.

CHAPTER

SIX

VISUALIZATIONS OVERVIEW

Performance profiles originate either from the user's own means (i.e. by building their own collectors and generating the profiles w.r.t *Specification of Profile Format*) or using one of the collectors from Perun's tool suite.

Perun can can interpret the profiling data in several ways:

1. By **directly running interpretation modules** through `perun show` command, that takes the profile w.r.t. *Specification of Profile Format* and uses various output backends (e.g. Bokeh, ncurses or plain terminal). The output method and format is up to the authors.
2. By **using python interpreter** together with internal modules for manipulation, conversion and querying the profiles (refer to *Profile API*, *Profile Query API*, and *Profile Conversions API*) and external statistical libraries, like e.g. using pandas.

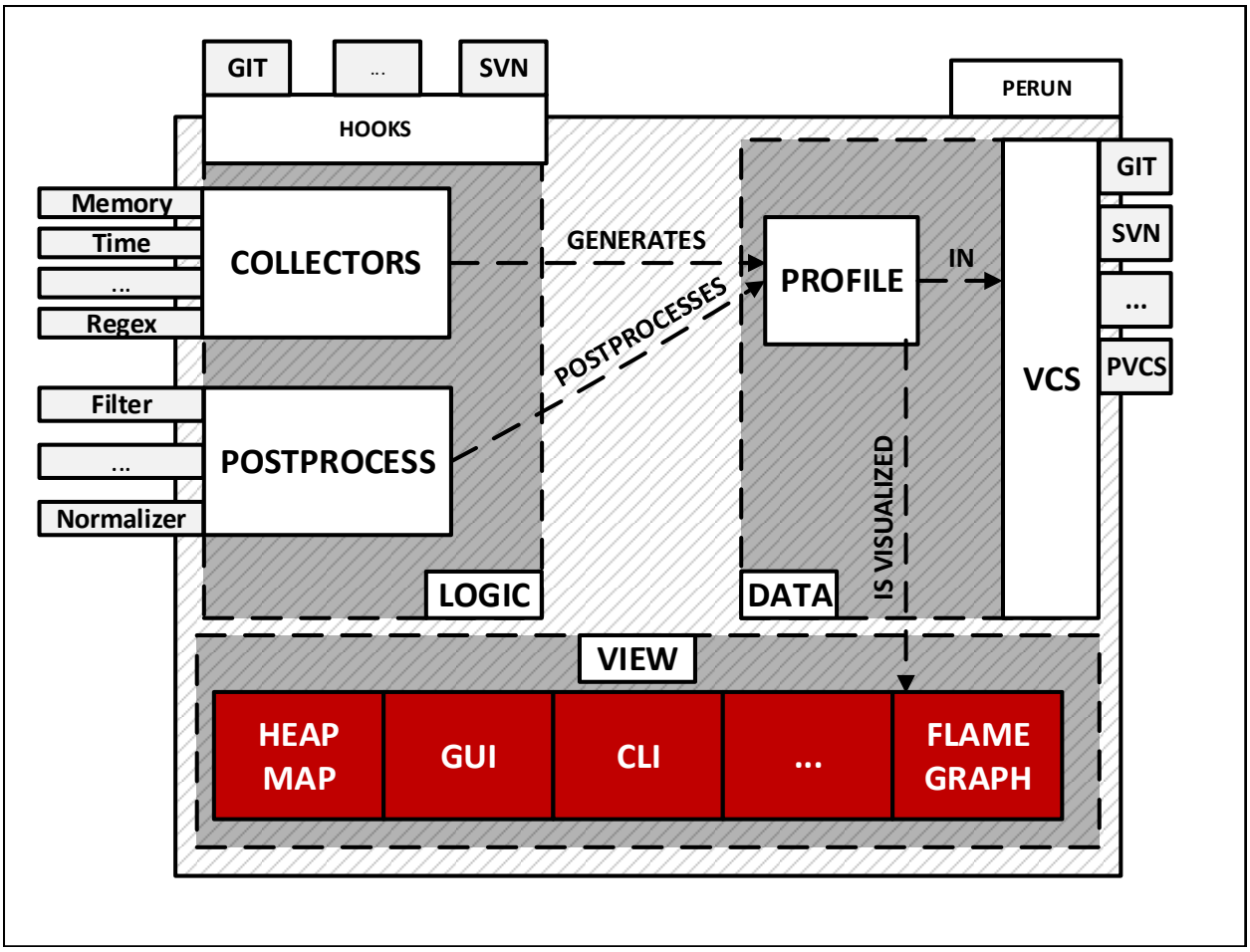
The format of input profiles has to be w.r.t. *Specification of Profile Format*, in particular the intepreted profiles should contain the `resources` region with data.

Automatically generated profiles are stored in the `.perun/jobs/` directory as a file with the `.perf` extension. The filename is by default automatically generated according to the following template:

```
bin-collector-workload-timestamp.perf
```

Refer to *Command Line Interface*, *Automating Runs*, *Collectors Overview* and *Postprocessors Overview* for more details about running command line commands, generating batch of jobs, capabilities of collectors and postprocessors techniques respectively. Internals of perun storage is described in *Perun Internals*.

Note that interface of show allows one to use *index* and *pending* tags of form `i@i` and `i@p` respectively, which serve as a quality-of-life feature for easy specification of visualized profiles.



6.1 Supported Visualizations

Perun's tool suite currently contains the following visualizations:

1. *Bars Plot* visualizes the data as bars, with moderate customization possibilities. The output is generated as an interactive HTML file using the Bokeh library, where one can e.g. move or resize the graph. *Bars* supports high number of profile types.
2. *Flow Plot* visualizes the data as flow (i.e. classical continuous graph), with moderate customization possiblities. The output is generated as an interactive HTML file using the Bokeh library, where one can move and resize the graph. *Flow* supports high number of profile types.
3. *Flame Graph* is an interface for Perl script of Brendan Gregg, that converts the (currently limited to memory profiles) profile to an internal format and visualize the resources as stacks of portional resource consumption depending on the trace of the resources.
4. *Scatter Plot* visualizes the data as points on two dimensional grid, with moderate customization possibilities. This visualization also display regression models, if the input profile was postprocessed by *Regression Analysis*.
5. *Heap Map* visualizes the *memory* consumption as a heap map of allocation resources to target memory addresses. Note that the output is dependent on ncurses library and hence can currently be used only from UNIX terminals.

All of the listed visualizations can be run from command line. For more information about command line interface for individual visualization either refer to *Collect units* or to corresponding subsection of this chapter.

For a brief tutorial how to create your own visualization module and register it in Perun for further usage refer to *Creating your own Visualization*. The format and the output is of your choice, it only has to be built over the format as described in *Specification of Profile Format* (or can be based over one of the conversions, see *Profile Conversions API*).

6.1.1 Bars Plot

Bar graphs displays resources as bars, with moderate customization possibilities (regarding the sources for axes, or grouping keys). The output backend of *Bars* is both Bokeh and ncurses (with limited possibilities though). Bokeh graphs support either the stacked format (bars of different groups will be stacked on top of each other) or grouped format (bars of different groups will be displayed next to each other).

Overview and Command Line Interface

perun show bars

Customizable interpretation of resources using the bar format.

- **Limitations**: *none*.
- **Interpretation style**: graphical
- **Visualization backend**: Bokeh

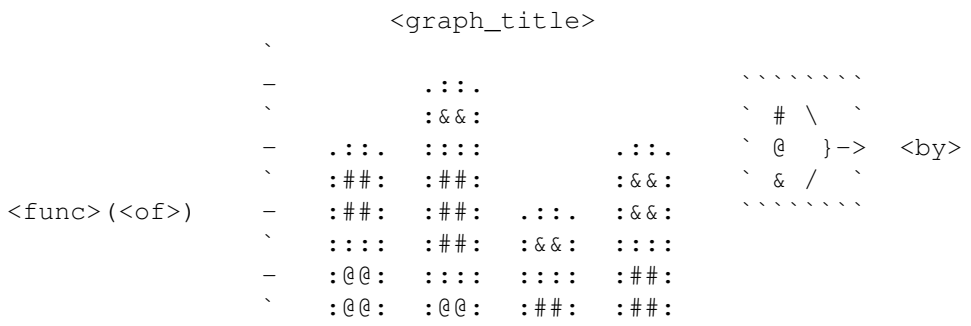
Bars graph shows the aggregation (e.g. sum, count, etc.) of resources of given types (or keys). Each bar shows `<func>` of resources from `<of>` key (e.g. sum of amounts, average of amounts, count of types, etc.) per each `<per>` key (e.g. per each snapshot, or per each type). Moreover, the graphs can either be (i) stacked, where the different values of `<by>` key are shown above each other, or (ii) grouped, where the different values of `<by>` key are shown next to each other. Refer to `resources` for examples of keys that can be used as `<of>`, `<key>`, `<per>` or `<by>`.

Bokeh library is the current interpretation backend, which generates HTML files, that can be opened directly in the browser. Resulting graphs can be further customized by adding custom labels for axes, custom graph title or different graph width.

Example 1. The following will display the sum of sum of amounts of all resources of given for each subtype, stacked by uid (e.g. the locations in the program):

```
perun show 0@i bars sum --of 'amount' --per 'subtype' --stacked --by 'uid'
```

The example output of the bars is as follows:



```
+````||````||````||````||````

<per>
```

Refer to *Bars Plot* for more thorough description and example of *bars* interpretation possibilities.

```
perun show bars [OPTIONS] <aggregation_function>
```

Options

-o, --of <of_key>
Sets key that is source of the data for the bars, i.e. what will be displayed on Y axis. [required]

-p, --per <per_key>
Sets key that is source of values displayed on X axis of the bar graph.

-b, --by <by_key>
Sets the key that will be used either for stacking or grouping of values

-s, --stacked
Will stack the values by <resource_key> specified by option –by.

-g, --grouped
Will stack the values by <resource_key> specified by option –by.

-f, --filename <filename>
Sets the outputs for the graph to the file.

-xl, --x-axis-label <x_axis_label>
Sets the custom label on the X axis of the bar graph.

-yl, --y-axis-label <y_axis_label>
Sets the custom label on the Y axis of the bar graph.

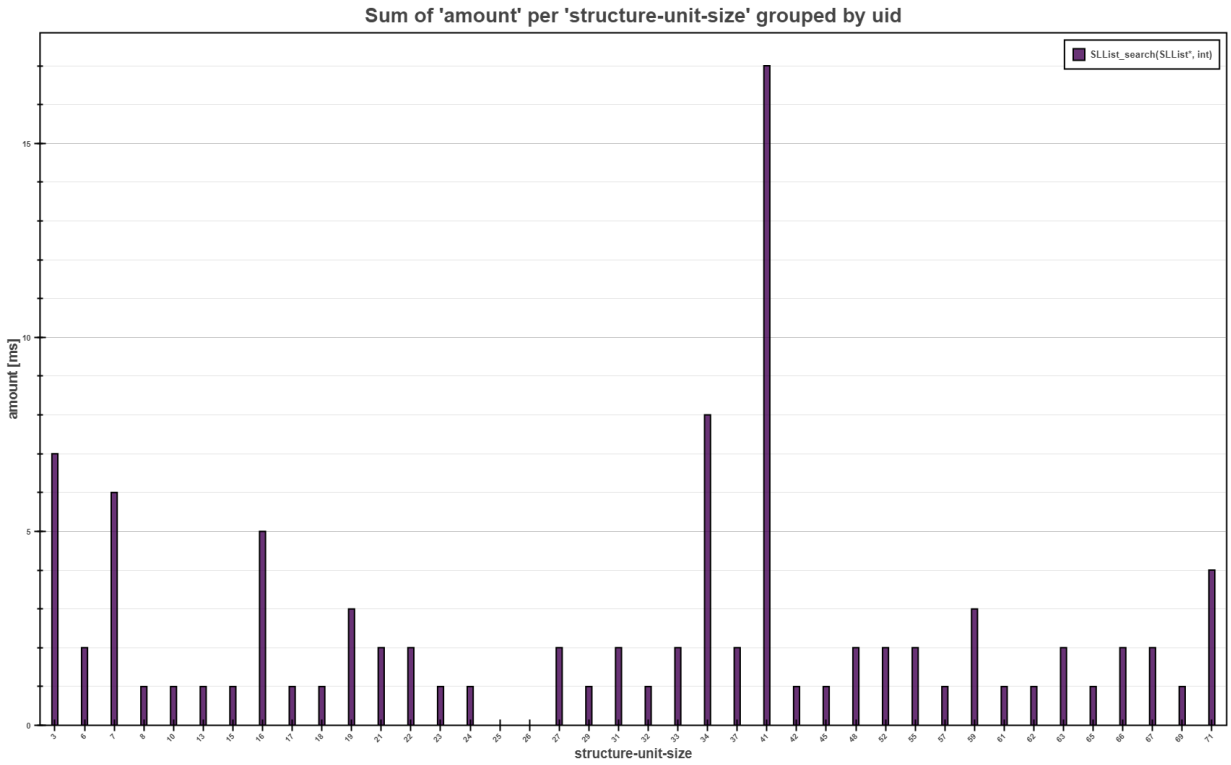
-gt, --graph-title <graph_title>
Sets the custom title of the bars graph.

-v, --view-in-browser
The generated graph will be immediately opened in the browser (firefox will be used).

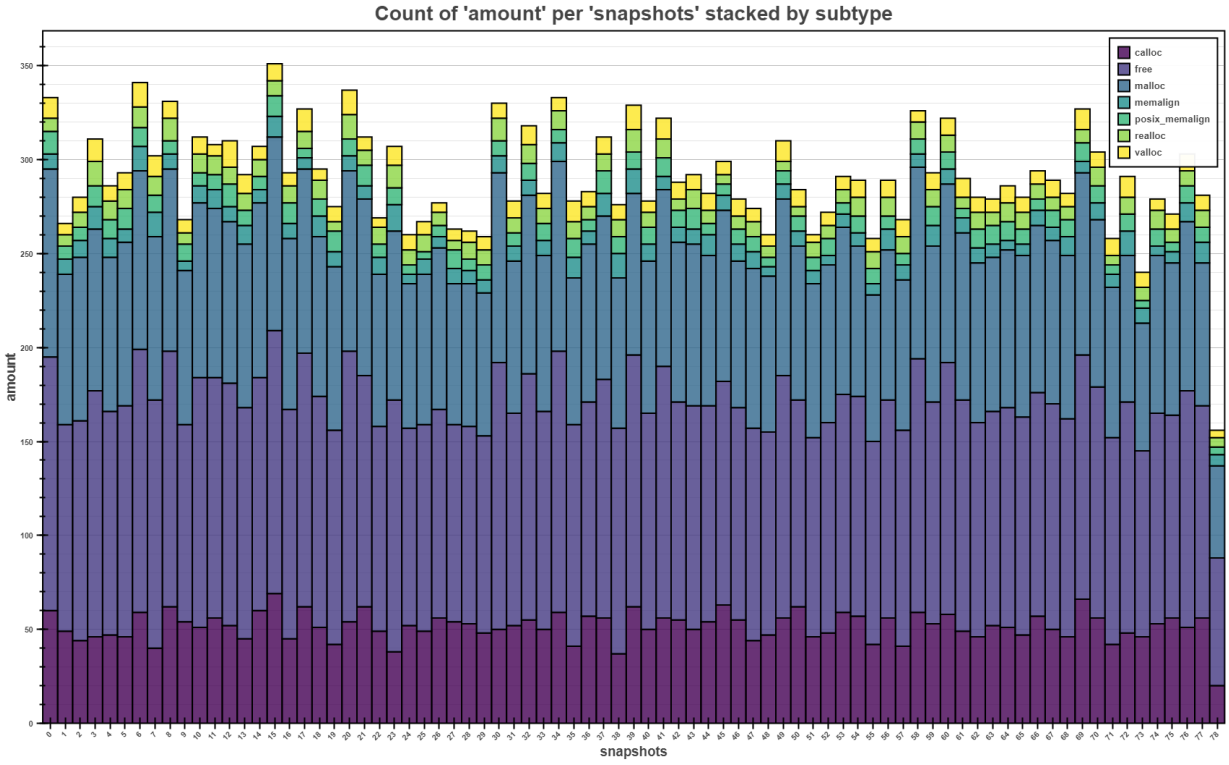
Arguments

<aggregation_function>
Optional argument

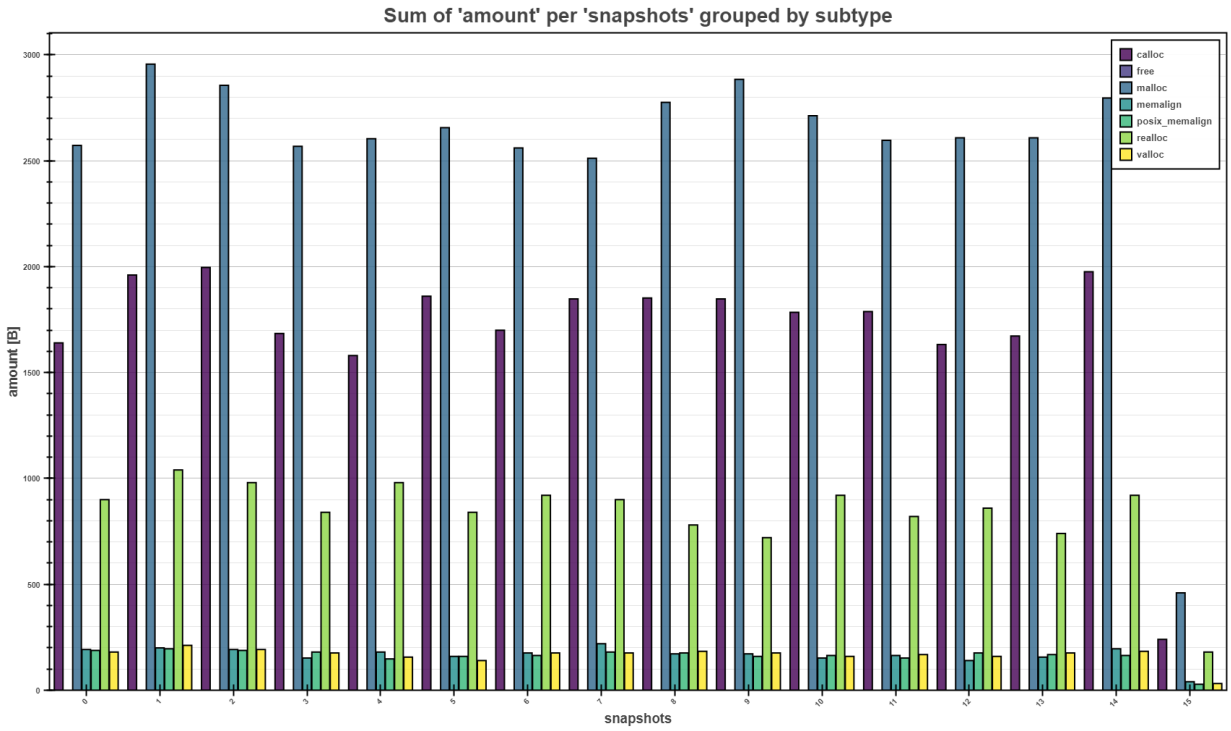
Examples of Output



The *Bars Plot* above shows the overall sum of the running times for each `structure-unit-size` for the `SLList_search` function collected by *Complexity Collector*. The interpretation highlights that the most of the consumed running time were over the single linked lists with 41 elements.



The *bars* above shows the *stacked* view of number of memory allocations made per each snapshot (with sampling of `1` second). Each bar shows overall number of memory operations, as well as proportional representation of different types of memory (de)allocation. It can also be seen that `free` is called approximately the same time as allocations, which signifies that everything was probably freed.



The *bars* above shows the *grouped* view of sum of memory allocation of the same type per each snapshot (with sampling of `0.5` seconds). Grouped pars allows fast comparison of total amounts between different types. E.g. `malloc` seems to allocated the most memory per each snapshot.

6.1.2 Flame Graph

Flame graph shows the relative consumption of resources w.r.t. to the trace of the resource origin. Currently it is limited to *memory* profiles (however, the generalization of the module is in plan). The usage of flame graphs is for faster localization of resource consumption hot spots and bottlenecks.

Overview and Command Line Interface

perun show flamegraph

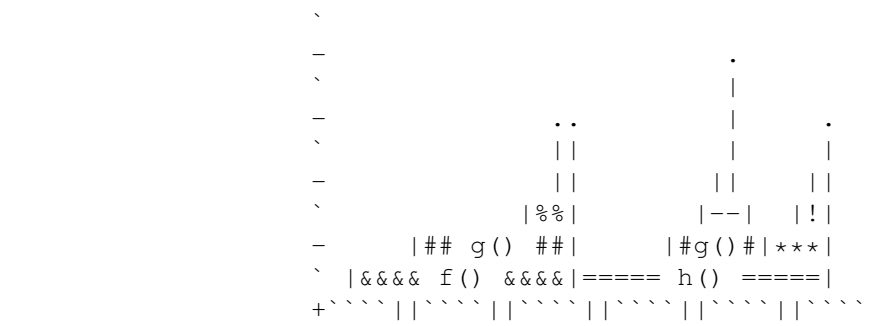
Flame graph interprets the relative and inclusive presence of the resources according to the stack depth of the origin of resources.

- **Limitations**: *memory* profiles generated by *Memory Collector*.
- **Interpretation style**: graphical
- **Visualization backend**: HTML

Flame graph intends to quickly identify hotspots, that are the source of the resource consumption complexity. On X axis, a relative consumption of the data is depicted, while on Y axis a stack depth is displayed. The wider the bars are on the X axis are, the more the function consumed resources relative to others.

Acknowledgements: Big thanks to Brendan Gregg for creating the original perl script for creating flame graphs w.r.t simple format. If you like this visualization technique, please check out this guy's site (http://brendangregg.com) for more information about performance, profiling and useful talks and visualization techniques!

The example output of the flamegraph is more or less as follows:



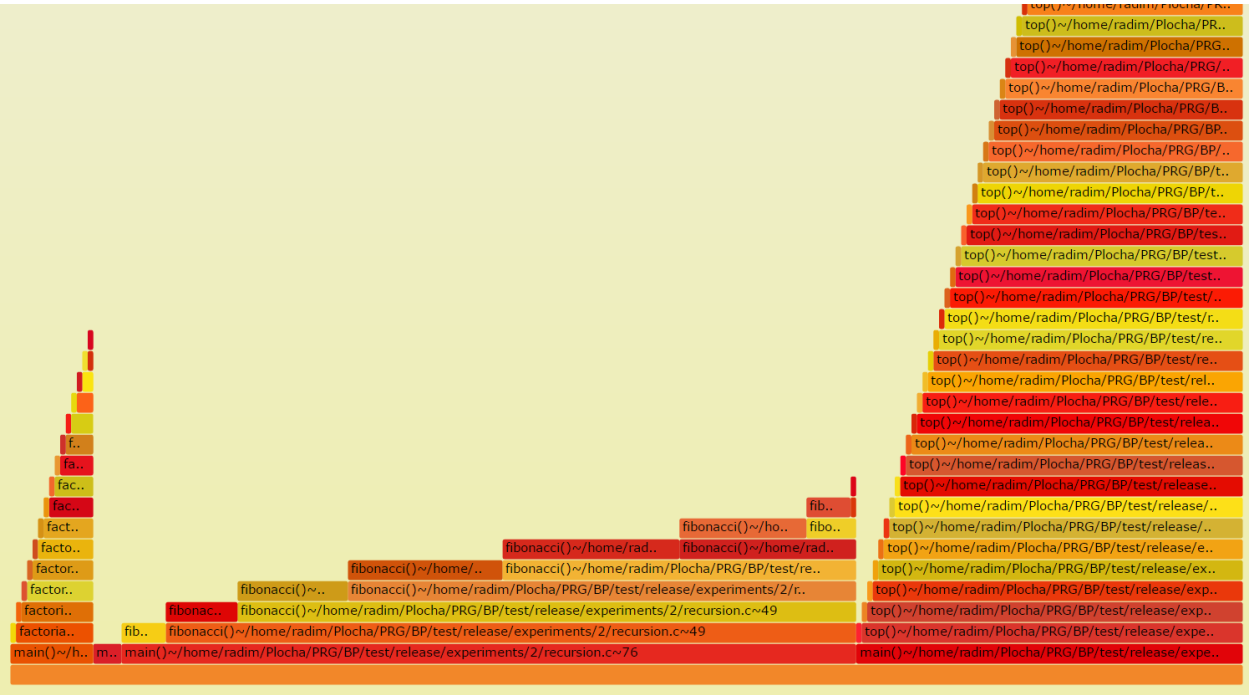
Refer to *Flame Graph* for more thorough description and examples of the interpretation technique. Refer to `perun.profile.convert.to_flame_graph_format()` for more details how the profiles are converted to the flame graph format.

```
perun show flamegraph [OPTIONS]
```

Options

-f, --filename <filename>
: Sets the output file of the resulting flame graph.

-h, --graph-height <graph_height>
: Increases the width of the resulting flame graph.



The *Flame Graph* is an efficient visualization of inclusive consumption of resources. The width of the base of one flame shows the bottleneck and hotspots of profiled binaries.

Examples of Output

6.1.3 Flow Plot

Flow graphs displays resources as classic plots, with moderate customization possibilities (regarding the sources for axes, or grouping keys). The output backend of *Flow* is both Bokeh and ncurses (with limited possibilities though). Bokeh graphs support either the classic display of resources (graphs will overlap) or in stacked format (graphs of different groups will be stacked on top of each other).

Overview and Command Line Interface

perun show flow

Customizable interpretation of resources using the flow format.

- **Limitations**: *none*.
- **Interpretation style**: graphical, textual
- **Visualization backend**: Bokeh, ncurses

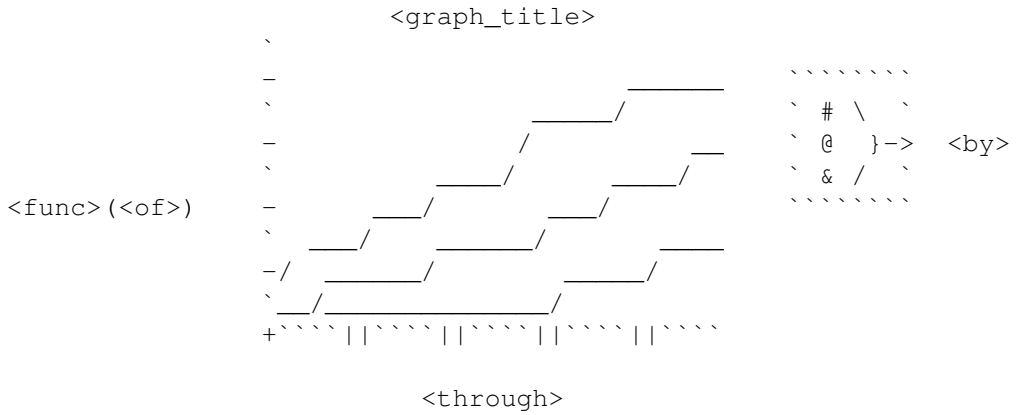
Flow graph shows the values resources depending on the independent variable as basic graph. For each group of resources identified by unique value of `<by>` key, one graph shows the dependency of `<of>` values aggregated by `<func>` depending on the `<through>` key. Moreover, the values can either be accumulated (this way when displaying the value of 'n' on x axis, we accumulate the sum of all values for all m < n) or stacked, where the graphs are output on each other and then one can see the overall trend through all the groups and proportions between each of the group.

Bokeh library is the current interpretation backend, which generates HTML files, that can be opened directly in the browser. Resulting graphs can be further customized by adding custom labels for axes, custom graph title or different graph width.

Example 1. The following will show the average amount (in this case the function running time) of each function depending on the size of the structure over which the given function operated:

```
perun show 0@i flow mean --of 'amount' --per 'structure-unit-size'
    --acumulated --by 'uid'
```

The example output of the bars is as follows:



Refer to *Flow Plot* for more thorough description and example of *flow* interpretation possibilities.

```
perun show flow [OPTIONS] <aggregation_function>
```

Options

-o, --of <of_key>
Sets key that is source of the data for the flow, i.e. what will be displayed on Y axis, e.g. the amount of resources. [required]

-t, --through <through_key>
Sets key that is source of the data value, i.e. the independent variable, like e.g. snapshots or size of the structure.

-b, --by <by_key>
For each <by_resource_key> one graph will be output, e.g. for each subtype or for each location of resource. [required]

-s, --stacked
Will stack the y axis values for different <by> keys on top of each other. Additionaly shows the sum of the values.

--accumulate, --no-accumulate
Will accumulate the values for all previous values of X axis.

-ut, --use-terminal
Shows flow graph in the terminal using ncurses library.

-f, --filename <filename>
Sets the outputs for the graph to the file.

-xl, --x-axis-label <x_axis_label>
Sets the custom label on the X axis of the flow graph.

-yl, --y-axis-label <y_axis_label>
Sets the custom label on the Y axis of the flow graph.

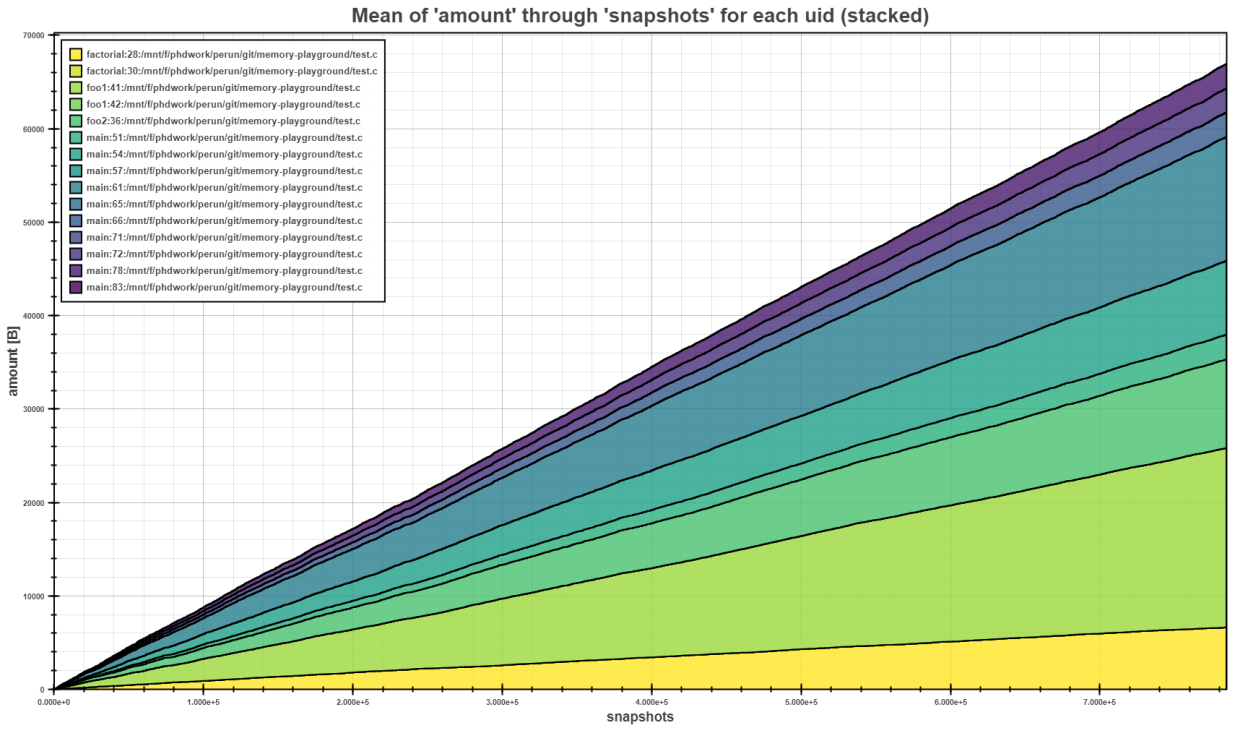
-gt, --graph-title <graph_title>
Sets the custom title of the flow graph.

-v, --view-in-browser
The generated graph will be immediately opened in the browser (firefox will be used).

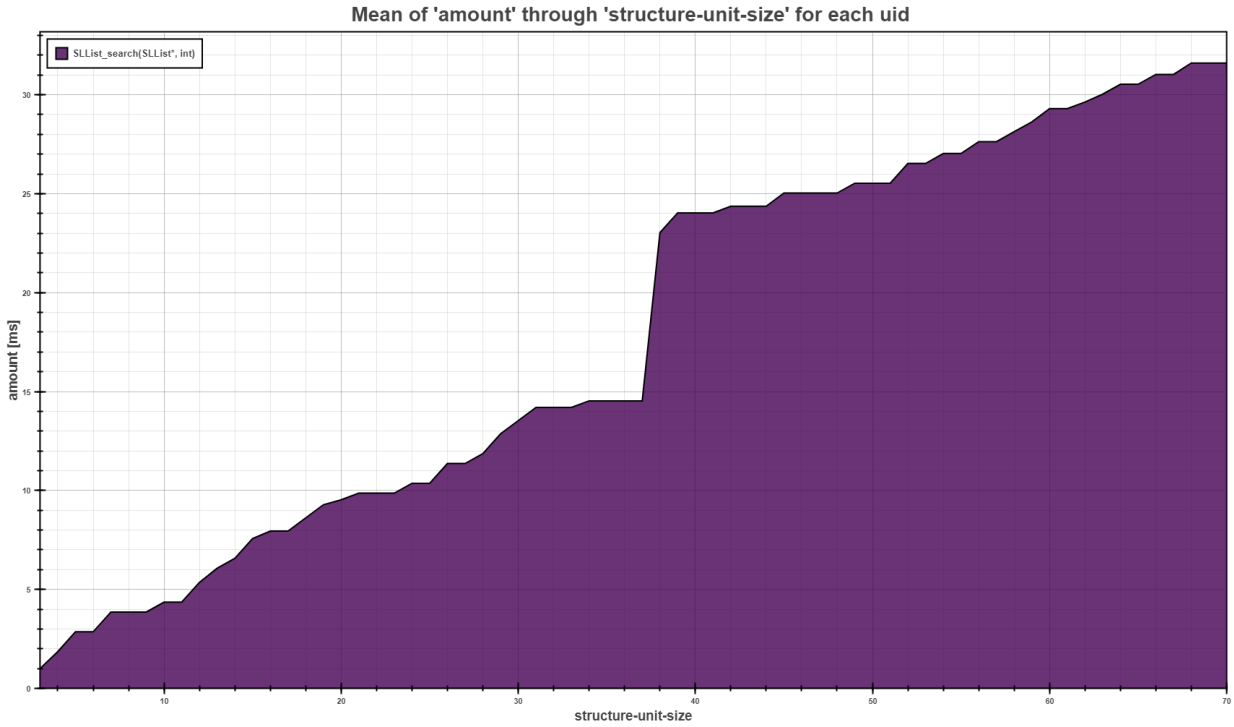
Arguments

<aggregation_function>
Optional argument

Examples of Output



The *Flow Plot* above shows the mean of allocated amounts per each allocation site (i.e. `uid`) in stacked mode. The stacking of the means clearly shows, where the biggest allocations where made during the program run.



The *Flow Plot* above shows the trend of the average running time of the `SLList_search` function depending on the size of the structure we execute the search on.

6.1.4 Heap Map

Heap map is a visualization of underlying memory address map that links chunks of allocated memory to corresponding allocation sources. This is mostly for showing utilization of memory, where objects were allocated, how often, and how the objects are fragmented in the memory. Heap map visualization is interactive and is implemented using ncurses library.

Overview and Command Line Interface

perun show heapmap

Shows interactive map of memory allocations to concrete memories for each function.

- **Limitations**: *memory* profiles generated by *Memory Collector*.
- **Interpretation style**: textual
- **Visualization backend**: ncurses

Heap map shows the underlying memory map, and links the concrete allocations to allocated addresses for each snapshot. The map is interactive, one can either play the full animation of the allocations through snapshots or move and explore the details of the map.

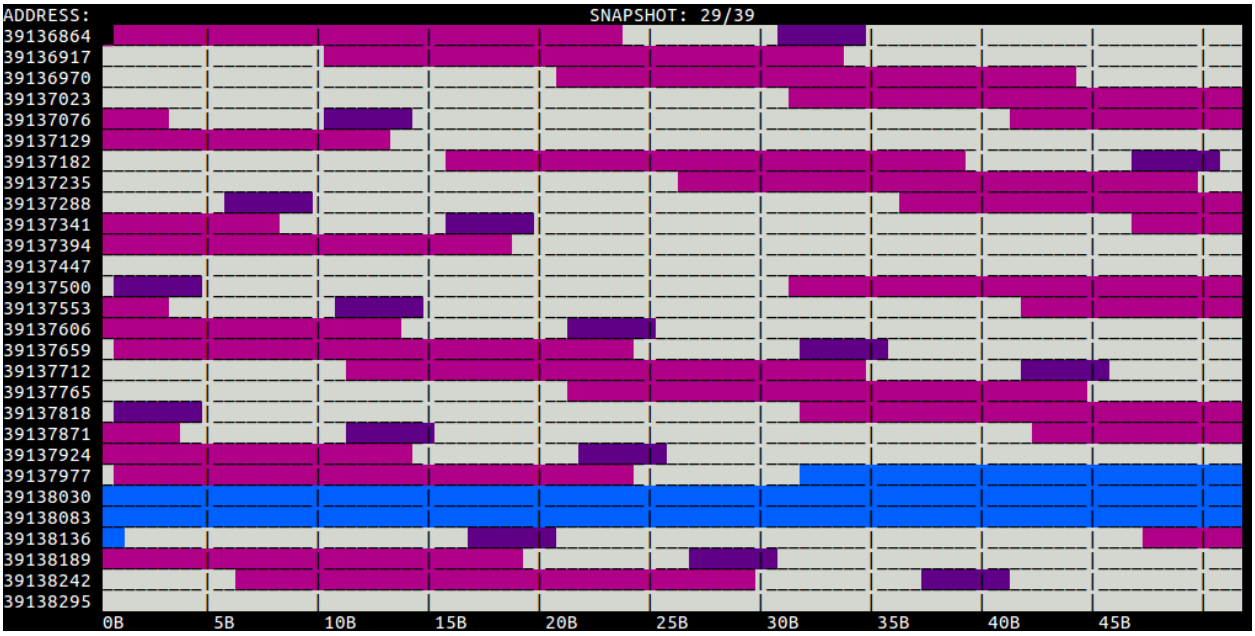
Moreover, the heap map contains *heat map* mode, which accumulates the allocations into the heat representation—the hotter the colour displayed at given memory cell, the more time it was allocated there.

The heap map aims at showing the fragmentation of the memory and possible differences between different allocation strategies. On the other hand, the heat mode aims at showing the bottlenecks of allocations.

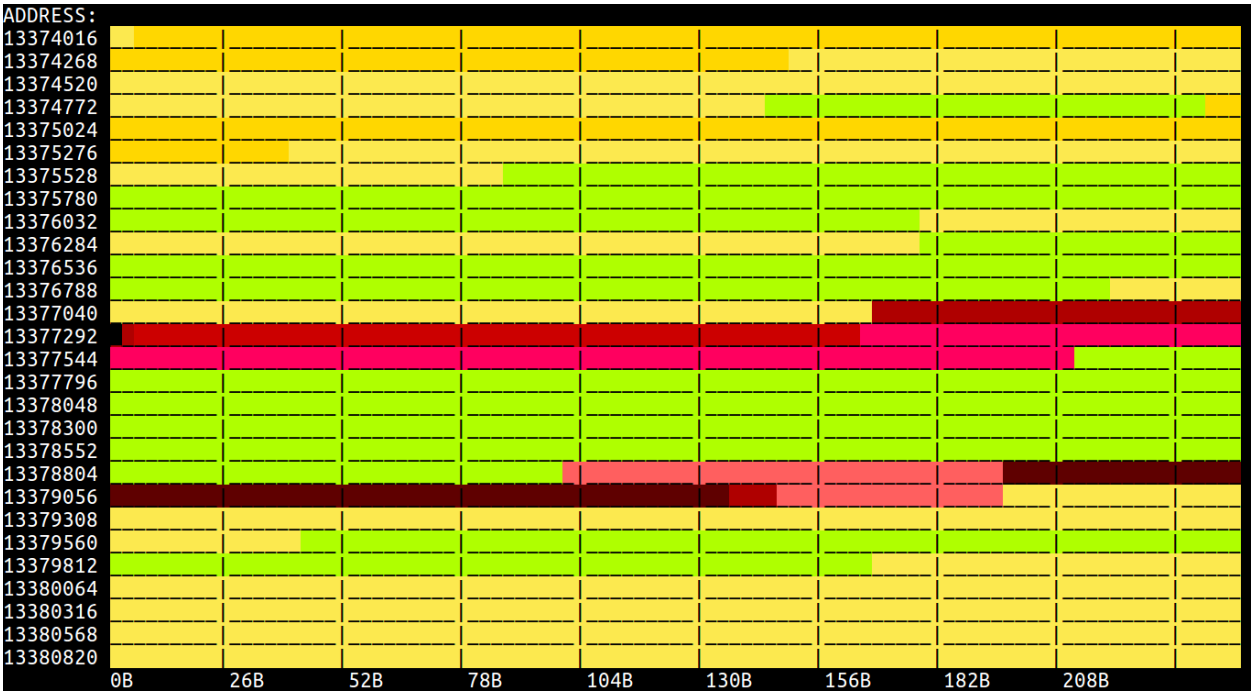
Refer to *Heap Map* for more thorough description and example of *heapmap* interpretation possibilities.

```
perun show heapmap [OPTIONS]
```

Examples of Output



The *Heap Map* shows the address space through the time (snapshots) and visualize the fragmentation of memory allocation per each allocation site. The *heap map* aboe shows the difference between allocations using lists (purple), skiplists (pinkish) and standard vectors (blue). The map itself is interactive and displays details about individual address cells.



Heat map is a mode of heap map, which aggregates the allocations over all of the snapshots and uses warmer colours for address cells, where more allocations were performed.

6.1.5 Scatter Plot

Scatter plot visualizes the data as points on two dimensional grid, with moderate customization possibilities. This visualization also display regression models, if the input profile was postprocessed by *Regression Analysis*. The output backend of *Scatter plot* is Bokeh library.

Overview and Command Line Interface

perun show scatter

Interactive visualization of resources and models in scatter plot format.

Scatter plot shows resources as points according to the given parameters. The plot interprets <per> and <of> as x, y coordinates for the points. The scatter plot also displays models located in the profile as a curves/lines.

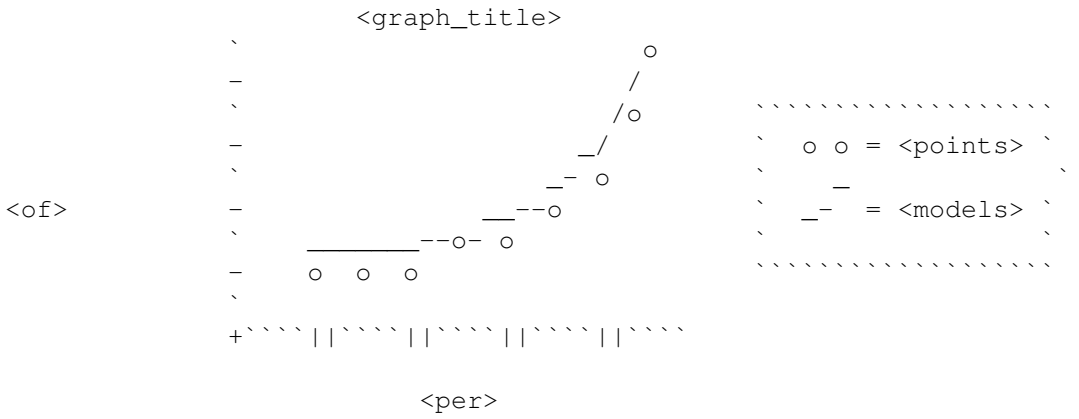
- **Limitations**: *none*.
- **Interpretation style**: graphical
- **Visualization backend**: Bokeh

Features in progress:

- uid filters
- models filters
- multiple graphs interpretation

Graphs are displayed using the Bokeh library and can be further customized by adding custom labels for axis, custom graph title and different graph width.

The example output of the scatter is as follows:



Refer to *Scatter Plot* for more thorough description and example of *scatter* interpretation possibilities. For more thorough explanation of regression analysis and models refer to *Regression Analysis*.

```
perun show scatter [OPTIONS]
```

Options

-o, --of <of_key>
Data source for the scatter plot, i.e. what will be displayed on Y axis. [default: amount]

-p, --per <per_key>
Keys that will be displayed on X axis of the scatter plot. [default: structure-unit-size]

-f, --filename <filename>
Outputs the graph to the file specified by filename.

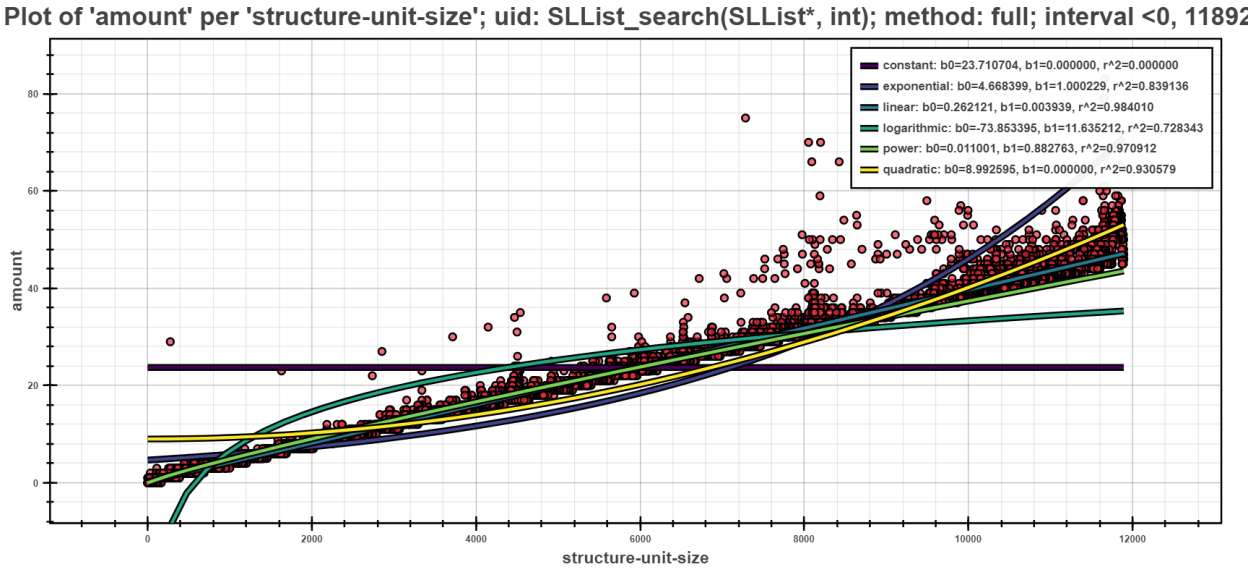
-xl, --x-axis-label <x_axis_label>
Label on the X axis of the scatter plot.

-yl, --y-axis-label <y_axis_label>
Label on the Y axis of the scatter plot.

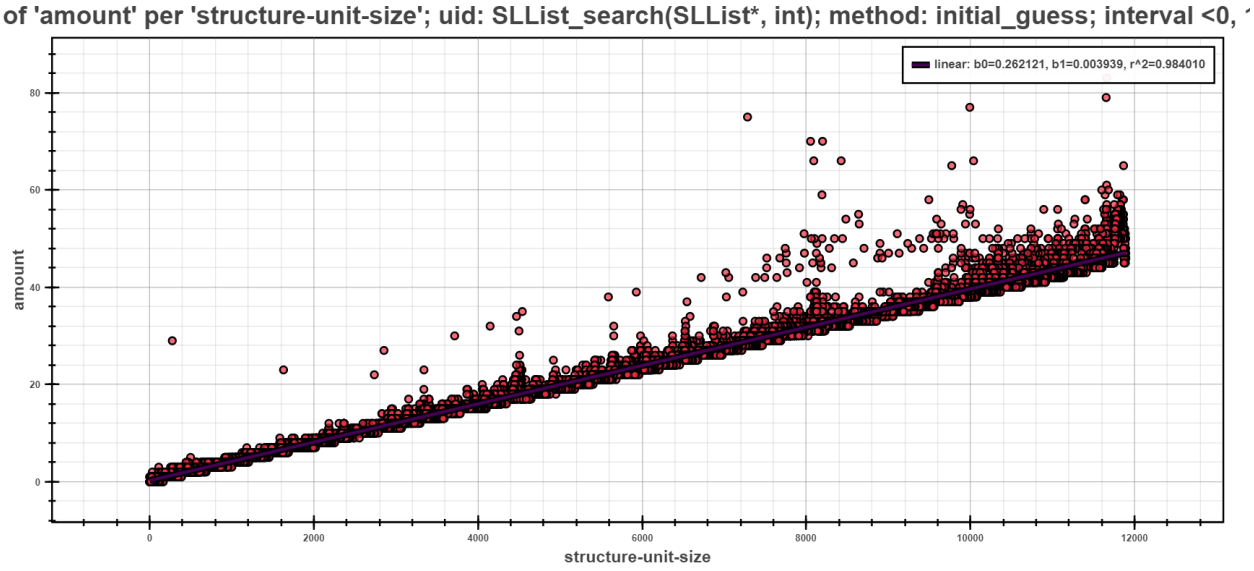
-gt, --graph-title <graph_title>
Title of the scatter plot.

-v, --view-in-browser
Will show the graph in browser.

Examples of Output



The *Scatter Plot* above shows the interpreted models of different complexity example, computed using the **full computation** method. In the picture, one can see that the depedency of running time based on the structural size is best fitted by *linear* models.



The next *scatter plot* displays the same data as previous, but regressed using the *initial guess* strategy. This strategy first does a computation of all models on small sample of data points. Such computation yields initial estimate of fitness of models (the initial sample is selected by random). The best fitted model is then chosen and fully computed on the rest of the data points.

The picture shows only one model, namely *linear* which was fully computed to best fit the given data points. The rest of the models had worse estimation and hence was not computed at all.

6.2 Creating your own Visualization

New interpretation modules can be registered within Perun in several steps. The visualization methods has the least requirements and only needs to work over the profiles w.r.t. *Specification of Profile Format* and implement method for Click api in order to be used from command line.

You can register your new visualization as follows:

1. Run `perun utils create view myview` to generate a new modules in `perun/view` directory with the following structure. The command takes a predefined templates for new visualization techniques and creates `__init__.py` and `run.py` according to the supplied command line arguments (see *Utility Commands* for more information about interface of `perun utils create` command):

```
/perun
|-- /view
    |-- /myview
        |-- __init__.py
        |-- run.py
    |-- /bars
    |-- /flamegraph
    |-- /flow
    |-- /heapmap
    |-- /scatter
```

2. First, implement the `__init__.py` file, including the module docstring with brief description of the visualization technique and definition of constants which has the following structure:

```
"""..."""

SUPPORTED_PROFILES = ['mixed|memory|mixed']

__author__ = 'You!'
```

3. Next, in the `run.py` implement module with the command line interface function, named the same as your visualization technique. This function is called from the command line as `perun show ``perun show myview` and is based on Click library.

4. Finally register your newly created module in `get_supported_module_names()` located in `perun.utils.__init__.py`:

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪views.py
@@ -8,5 +8,6 @@
         'vcs': ['git'],
         'collect': ['complexity', 'memory', 'time'],
         'postprocess': ['filter', 'normalizer', 'regression_analysis'],
-        'view': ['alloclist', 'bars', 'flamegraph', 'flow', 'heapmap', 'raw
↪', 'scatter']
+        'view': ['alloclist', 'bars', 'flamegraph', 'flow', 'heapmap', 'raw
↪', 'scatter',
+                 'myview']
     }[package]
```

5. Preferably, verify that registering did not break anything in the Perun and if you are not using the developer installation, then reinstall Perun:

```
make test
make install
```

6. At this point you can start using your visualization either using `perun show`.

7. If you think your collector could help others, please, consider making Pull Request.

CHAPTER

SEVEN

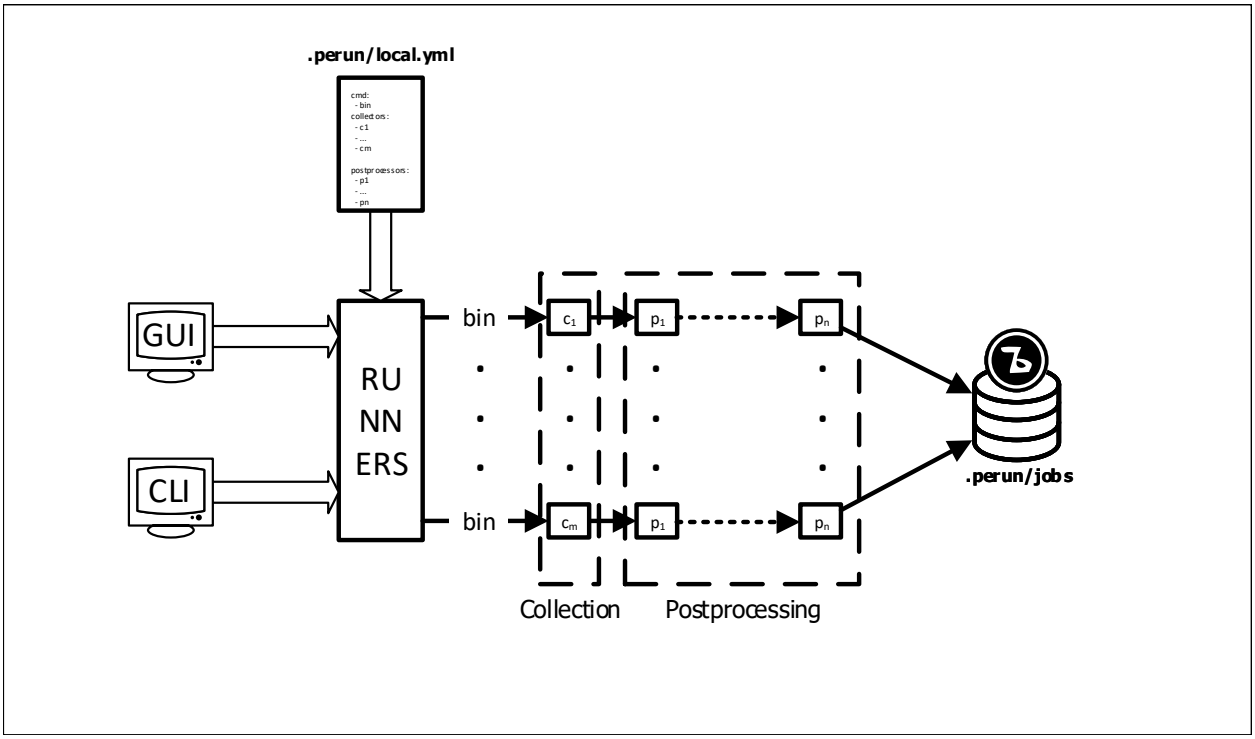
AUTOMATING RUNS

Profiles can be generated either manually on your own (either by individual profilers or using the `perun collect` and `perun postprocess` commands), or you can use Perun's `runner` infrastructure to partially automate the generation process. Perun is capable either to run the jobs through the stored configuration which is meant for a regular project profiling (either in local or shared configuration c.f. *Perun Configuration files*) or through a single job specifications meant for irregular or specific profiling jobs.

Each profile generated by specified batch jobs will be stored in `.perun/jobs` directory with the following name of the template:

```
command-collector-workload-Y-m-d-H-M-S.perf
```

Where `command` corresponds to the name of the application (or script), for which we collected the data using `collector` on `workload` at given specified date. You can change the template for profile name generation by setting *`format.output_profile_template`*. New profiles are annotated with the *`origin`* set to the current `HEAD` of the wrapped repository. *origin* serves as a check during registering profiles in the indexes of minor versions. Profile with *origin* different from the target minor version will not be assigned, as it would violate the correctness of the performance history of the project.



The figure above show the overview of the jobs flow in Perun. The runner module is initialized form user interfaces and from local (or shared) configurations and internally generates the matrix of jobs which are run in the sequence. Each job is then finished with storing the generated profile in the internal storage.

Note: In order to obtain fine result, it is advised to run the benchmark several times (at least three times) and either do the average over all runs or discard the first runs. This is because, initial benchmarks usually have skewed times.

Note: If you do not want to miss profiling, e.g. after each push, commit, etc., check out git hooks. `git hooks` allows you to run custom scripts on certain git event triggers.

7.1 Runner CLI

Command Line Interface contains group of two commands for managing the jobs—`perun run job` for running one specified batch of jobs (usually corresponding to irregular measuring or profilings) and `perun run matrix` for running the pre-configured matrix in Yaml format specifing the batch job (see *Job Matrix Format* for full specification). Running the jobs by `perun run matrix` corresponds to regular measuring and profiling, e.g. during end of release cycles, before push to origin/upstream or even after each commit.

7.1.1 perun run job

Run specified batch of perun jobs to generate profiles.

This command correspond to running one isolated batch of profiling jobs, outside of regular profilings. Run `perun run matrix`, after specifying job matrix in local configuration to automate regular profilings of your project. After the batch is generated, each profile is taged with *`origin`* set to current `HEAD`. This serves as check to not assing such profiles to different minor versions.

By default the profiles computed by this batch job are stored inside the `.perun/jobs/` directory as a files in form of:

```
bin-collector-workload-timestamp.perf
```

In order to store generated profiles run the following, with `i@p` corresponding to *pending tag*, which can be obtained by running `perun status`:

```
perun add i@p
```

```
perun run job -c time -b ./mybin -w file.in -w file2.in -p normalizer
```

This command profiles two commands `./mybin file.in` and `./mybin file2.in` and collects the profiling data using the *Time Collector*. The profiles are afterwards normalized with the *Normalizer Postprocessor*.

```
perun run job -c complexity -b ./mybin -w sll.cpp -cp complexity targetdir=./src
```

This commands runs one job './mybin sll.cpp' using the *Complexity Collector*, which uses custom binaries targeted at `./src` directory.

```
perun run job -c mcollect -b ./mybin -b ./otherbin -w input.txt -p normalizer -p␣
↪regression_analysis
```

This commands runs two jobs `./mybin input.txt` and `./otherbin input.txt` and collects the profiles using the *Memory Collector*. The profiles are afterwards postprocessed, first using the *Normalizer Postprocessor* and then with *Regression Analysis*.

Refer to *Automating Runs* and *Perun's Profile Format* for more details about automation and lifetimes of profiles. For list of available collectors and postprocessors refer to *Supported Collectors* and *Supported Postprocessors* respectively.

```
perun run job [OPTIONS]
```

Options

-b, --cmd <cmd>
Command that is being profiled. Either corresponds to some script, binary or command, e.g. `./mybin` or `perun`. [required]

-a, --args <args>
Additional parameters for <cmd>. E.g. `status` or `-al` is command parameter.

-w, --workload <workload>
Inputs for <cmd>. E.g. `./subdir` is possible workloadfor `ls` command.

-c, --collector <collector>
Profiler used for collection of profiling data for the given <cmd> [required]

-cp, --collector-params <collector_params>
Additional parameters for the <collector> read from the file in YAML format

-p, --postprocessor <postprocessor>
After each collection of data will run <postprocessor> to postprocess the collected resources.

-pp, --postprocessor-params <postprocessor_params>
Additional parameters for the <postprocessor> read from the file in YAML format

7.1.2 perun run matrix

Runs the jobs matrix specified in the local.yml configuration.

This commands loads the jobs configuration from local configuration, builds the *job matrix* and subsequently runs the jobs collecting list of profiles. Each profile is then stored in `.perun/jobs` directory and moreover is annotated using by setting *`origin`* key to current `HEAD`. This serves as check to not assing such profiles to different minor versions.

The job matrix is defined in the yaml format and consists of specification of binaries with corresponding arguments, workloads, supported collectors of profiling data and postprocessors that alter the collected profiles.

Refer to *Automating Runs* and *Job Matrix Format* for more details how to specify the job matrix inside local configuration and to *Perun Configuration files* how to work with Perun's configuration files.

```
perun run matrix [OPTIONS]
```

7.2 Overview of Jobs

Usually during the profiling of application, we first collect the data by the means of profiler (or profiling data collector or whatever terminology we are using) and we can further augment the collected data by ordered list of postprocessing phases (e.g. for filtering out unwanted data, normalizing or scaling the amounts, etc.). As results we generate one

profile for each application configuration and each profiling job. Thus, we can consider one profiling jobs as collection of profiling data from application of one certain configuration using one collector and ordered set of postprocessors.

One configuration of application can be partitioned into three parts (two being optional):

1. The actual **command** that is being profiled, i.e. either the binary or wrapper script that is executed as one command from the terminal and ends with success or failure. An example of command can be e.g. the `perun` itself, `ls` or `./my_binary`.
2. Set of **arguments** for command (*optional*), i.e. set of parameters or arguments, that are supplied to the profiled command. The intuition behind arguments is to allow setting various optimization levels or profile different configurations of one program. An example of argument (or parameter) can be e.g. `log`, `-al` or `-O2 -v`.
3. Input **workloads** (*optional*), i.e. different inputs for profiled command. While workloads can be considered as arguments, separating them allows more finer specification of jobs, e.g. when we want to profile our program on workloads with different sizes under different configurations (since degradations usually manifest under bigger workloads). An example of workload can be e.g. `HEAD` or `/dir/subdir` or `<< "Hello world"`.

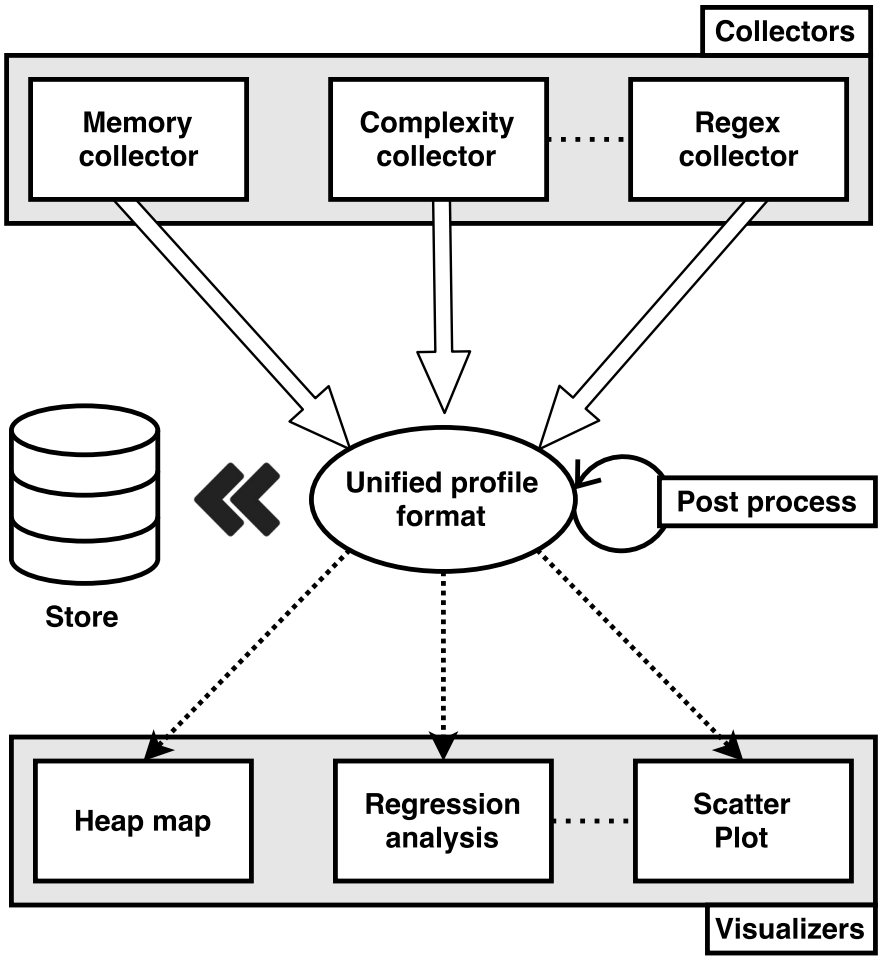
So from the user specification, commands, arguments and workloads can be combined using cartesian product which yields the list of full application configurations. Then for each such configuration (like e.g. `perun log HEAD`, `ls -al /dir/subdir` or `./my_binary -O2 -v << "Hello world"`) we run specified collectors and finally the list of postprocessors. This process is automatic either using the `perun run job` or `perun run matrix`, which differ in the way how the user specification is obtained.

Each collector (resp. postprocessor) runs in up to three phases (with *pre* and *post* phases being optional). First the function `before()` is executed (if implemented by given collector or postprocessor), where the collector (resp. postprocessor) can execute additional preparation before the actual collection (resp. postprocessing) of the data, like e.g. compiling custom binaries. Then the actual `collect()` (resp. `postprocess()`) is executed, which runs the given job with specified collection (resp. postprocessing) unit and generatesj profile (potentially in raw or intermediate format). Finally the `after()` phase is run, which can further postprocess the generated profile (after the success of collection), e.g. by required filtering of data or by transforming raw profiles to *Perun's Profile Format*. See (*Collectors Overview* and *Postprocessors Overview* for more detailed description of units). During these phases `kwargs` are passed through and share the specification, or can be used for passing additional information to following phases. The resulting `kwargs` has to contain the `profile` key, which contains the profile w.r.t. *Specification of Profile Format*.

The overall process can described by the following pseudocode:

```
for (cmd, argument, workload) in jobs:
   for collector in collectors:
      collector.before(cmd, argument, workload)
      collector.collect(cmd, argument, workload)
      profile = collector.after()
      for postprocessor in postprocessors:
         postprocessor.before(profile)
         postprocessor.postprocess(profile)
         profile = postprocessor.after(profile)
```

Note that each phase should return the following tripple: (`status code`, `status message`, `kwargs`). The `status code` is used for checking the success of the called phases and in case of error prints the `status message`.



For specification and details about collectors, postprocessors and internal storage of Perun refer to *Collectors Overview*, *Postprocessors Overview* and *Perun Internals*.

7.3 Job Matrix Format

In order to maximize the automation of running jobs you can specify in Perun config the specification of commands, arguments, workloads, collectors and postprocessors (and their internal configurations) as specified in the *Overview of Jobs*. *Job matrixes* are meant for a regular profiling jobs and should reduce the profiling to a single `perun run matrix` command. Both the config and the specification of job matrix is based on Yaml format.

Full example of one job matrix is as follows:

```
cmds:
   - perun

args:
   - log
   - log --short

workloads:
   - HEAD
   - HEAD~1
```

```
collectors:
   - name: time

postprocessors:
   - name: normalizer
   - name: regression_analysis
     params:
      - method: full
      - steps: 10
```

Given matrix will create four jobs (`perun log HEAD`, `perun log HEAD~1`, `perun log --short HEAD` and `perun log --short HEAD~1`) which will be issued for runs. Each job will be collected by *Time Collector* and then postprocessed first by *Normalizer Postprocessor* and then by *Regression Analysis* with specification `{'method': 'full', 'steps': 10}`.

Run the following to configure the job matrix of the current project:

```
perun config --edit
```

This will open the local configuration in editor specified by *general.editor* and lets you specify configuration for your application and set of collectors and postprocessors. Unless the source configuration file was not modified, it should contain a helper comments. The following keys can be set in the configuration:

cmds
: List of names of commands which will be profiled by set of collectors. The commands should preferably not contain any parameters or workloads, since they can be set by different configuration resulting into finer specification of configuration.

```
cmds:
   - perun
   - ls
   - ./myclientbinary
   - ./myserverbinary
```

args
: List of arguments (or parameters) which are supplied to profiled commands. It is advised to differentiate between arguments/parameters and workloads. While their semantics may seem close, separation of this concern results into more verbose performance history

```
args:
   - log
   - log --short
   - -al
   - -q -O2
```

workloads
: List of workloads which are supplied to profiled commands. Workloads represents program inputs and supplied files.

```
workloads:
   - HEAD
   - HEAD~1
   - /usr/share
   - << "Hello world!"
```

collectors
: List of collectors used to collect data for the given configuration of application represented by commands,

arguments and workloads. Each collector is specified by its *name* and additional *params* which corresponds to the dictionary of (key, value) parameters. Note that the same collector can be specified more than once (for cases, when one needs different collector configurations). For list of supported collectors refer to *Supported Collectors*.

```
collectors:
   - name: memory
     params:
         - sampling: 1
   - name: time
```

postprocessors

List of postprocessors which are used after the successful collection of the profiling data. Each postprocessor is specified by its *name* and additional *params* which corresponds to the dictionary of (key, value) parameters. Note that the same postprocessor can be specified more than just once. For list of supported postprocessors refer to *Supported Postprocessors*.

```
postprocessors:
   - name: normalizer
   - name: regression_analysis
     params:
      - method: full
      - steps: 10
```

CHAPTER

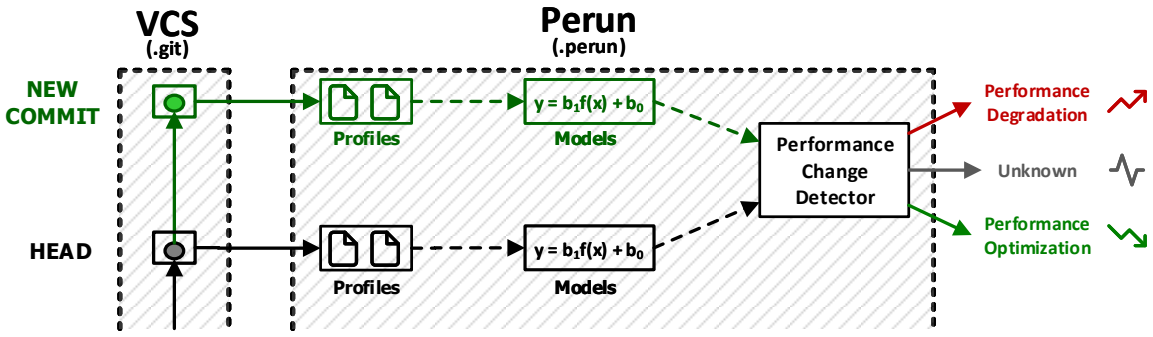
EIGHT

DETECTING PERFORMANCE CHANGES

For every new minor version of project (or every project release), developers should usually generate new batch of performance profiles with the same concrete configuration of resource collection (i.e. the set of collectors and postprocessors run on the same commands).These profiles are then assigned to the minor version to preserve the history of the project performance. However, every change of the project, and every new minor version, can cause a performance degradation of the project. And manual evaluation whether the degradation has happened is hard.

Perun allows one to automatically check the performance degradation between various minor versions within the history and protect the project against potential degradation introduced by new minor versions. One can employ multiple strategies for different configurations of profiles, each suitable for concrete types of degradation or performance bugs. Potential changes of performance are then reported for pairs of profiles, together with more precise information, such as the location, the rate or the confidence of the detected change. These information then help developer to evaluate whether the detected changes are real or spurious. The spurious warnings can naturally happen, since the collection of data is based on dynamic analysis and real runs of the program; and both of them can be influenced heavily by environment or other various aspects, such as higher processor utilization.

The detection of performance change is always checked between two profiles with the same configuration (i.e collected by same collectors, postprocessed using same postprocessors, and collected for the same combination of command, arguments and workload). These profiles correspond to some minor version (so called target) and its parents (so called baseline). But baseline profiles do not have to be necessarily the direct predecessor (i.e. the old head) of the target minor version, and can be found deeper in the version hierarchy (e.g. the root of the project or minor version from two days ago, etc.). During the check of degradation of one profile corresponding to the target, we find the nearest baseline profile in the history. Then for one pair of target and baseline profiles we can use multiple methods and these methods can then report multiple performance changes (such as optimizations and degradations).



8.1 Results of Detection

Between the pair of target and baseline profile one can use multiple methods, each suitable for specific type of change. Each such method can then yield multiple reports about detected performance changes (however, some of these can be spurious). Each degradation report can contain the following details:

1. **Type of the change**—the overall general classification of the performance change, which can be one of the following six values representing both certain and uncertain answers:

   `No Change`:

   > Represents that the performance of the given uniquely identified resource group was not changed in any way and it stayed the same (within some bound of error). By default these changes are not reported in the standard output, but can be made visible by increasing the verbosity of the command line interface (see *Command Line Interface* how to increase the verbosity of the output).

   `Degradation` or `Optimization`:

   > Represents that the performance of resource group has degradated (resp optimized), i.e. got worse (resp got better) with a fairly high confidence. Each report also usually shows the confidence of this report, e.g. by the value of coefficient of determination (see *Regression Analysis*), which quantifies how the prediction or regression models of both versions were fitting the data.

   `Maybe Degradation` or `Maybe Optimization`:

   > Represents detected performance change which is either unverified or with a low confidence (so the change can be either false positive or false negative). This classification of changes allows methods to provide more broader evaluation of performance change.

   `Unknown`:

   > Represents that the given method could not determine anything at all.

2. **Subtype of the change**—the description of the type of the change in more details, such as that the change was in *complexity order* (e.g. the performance model degraded from linear model to power model) or *ratio* (e.g. the average speed degraded two times)
3. **Confidence**—an indication how likely the degradation is real and not spurious or caused by badly collected data. The actual form of confidence is dependent on the underlying detection method. E.g. for methods based on *Regression Analysis* this can correspond to the coefficient of determination which shows the fitness of the function models to the actually measured values.
4. **Location**—the unique identification of the group of resources, such as the name of the function, the precise chunk of the code or line in code.

If the underlying method does not detect any change between two profiles, by default nothing is reported at all. However, this behaviour can be changed by increasing the verbosity of the output (see *Command Line Interface* how to increase the verbosity of the output)

8.2 Detection Methods

Currently we support two simple strategies for detection of the performance changes:

1. *Best Model Order Equality* which is based on results of *Regression Analysis* and only checks for each uniquely identified group of resources, whether the best performance (or prediction) model has changed (considering lexicographic ordering of model types), e.g. that the best model changed from *linear* to *quadratic*.

2. *Average Amount Threshold* which computes averages as a representation of the performance for each uniquely identified group of resources. Each average of the target is then compared with the average of the baseline and if the their ration exceeds a certain threshold interval, the method reports the change.

Refer to *Create Your Own Degradation Checker* to create your own detection method.

8.2.1 Best Model Order Equality

- **Detects**: *Order* changes; `Optimization` and `Degradation`
- **Confidence**: Minimal *coefficient of determination* of best models of baseline and target minor versions
- **Limitations**: Profiles postprocessed by *Regression Analysis*

The *Best Model Order Equality* choses the best model (i.e. the one with the highest *coefficient of determination*) as the representant of the performance of each group of uniquely identified resources (e.g. corresponding to the same function). Then each pair of baseline and target models is compared lexicographically (e.g. the *linear* model is lexicographically smaller than *quadratic* model), and any change in this ordering is detected as either `Optimization` or `Degradation` if the minimal confidence of the models is above certain threshold.

The example of the output generated by the *BMOE* method is as follows

```
* 1eb3d6: Fix the degradation of search
|\
| * 7813e3: Implement new version of search
|   > collected by complexity+regression_analysis for cmd: '$ mybin'
|     > applying 'best_model_order_equality' method
|       - Optimization         at SLList_search(SLList*, int)
|           from: power -> to: linear (with confidence r_square = 0.99)
|
* 7813e3: Implement new version of search
|\
| * 503885: Fix minor issues
|   > collected by complexity+regression_analysis for cmd: '$ mybin'
|     > applying 'best_model_order_equality' method
|       - Degradation          at SLList_search(SLList*, int)
|           from: linear -> to: power (with confidence r_square = 0.99)
|
* 503885: Fix minor issues
```

In the output above, we detected the `Optimization` between commits `1eb3d6` (target) and `7813e3` (baseline), where the best performance model of running time of `SLList_search` function changed from **power** model to **linear**. For the methods based on *Regression Analysis* we use the *coefficient of determination* (r^2) to represent a confidence, and take the minimal *coefficient of determination* of target and baseline model as a confidence for this detected change. Since r^2 is almost close to the value *1.0* (which would mean, that the model precisely fits the measured values), this signifies that the best model fit the data tightly and hence the detected optimization is **not spurious**.

8.2.2 Average Amount Threshold

- **Detects**: *Ratio* changes; `Optimization` and `Degradation`
- **Confidence**: *None*
- **Limitations**: *None*

The *Average Amount Threshold* groups all of the resources according to the unique identifier (uid; e.g. the function name) and then computes the averages of resource amounts as performance representants of baseline and target profiles. The computed averages are then compared (by division , and according to the set threshold the checker detects either `Optimization` or `Degradation` (the threshold is `2.0` ratio for detecting degradation and `0.5` ratio for detecting optimization, i.e. the threshold is two times speed-up or speed-down)

The example of output generated by *AAT* method is as follows:

```
* 1eb3d6: Fix the degradation of search
|\
| * 7813e3: Implement new version of search
|   > collected by complexity+regression_analysis for cmd: '$ mybin'
|     > applying 'average_amount_threshold' method
|       - Optimization         at SLList_search(SLList*, int)
|           from: 60677.98ms -> to: 135.29ms
|
* 7813e3: Implement new version of search
|\
| * 503885: Fix minor issues
|   > collected by complexity+regression_analysis for cmd: '$ mybin'
|     > applying 'average_amount_threshold' method
|       - Degradation          at SLList_search(SLList*, int)
|           from: 156.48ms -> to: 60677.98ms
|
* 503885: Fix minor issues
```

In the output above, we detected the `Optimization` between commits `1eb3d6` (target) and `7813e3` (baseline), where the average amount of running time for `SLList_search` function changed from about six seconds to hundred miliseconds. For these detected changes we report no confidence at all.

8.3 Configuring Degradation Detection

We apply concrete methods of performance change detection to concrete pairs of profiles according to the specified *rules* based on profile collection configuration. By *configuration* we mean the tuple of *(command, arguments, workload, collector, postprocessors)* which represent how the data were collected for the given minor version. This way for each new version of project, it is meaningful to collect new data using the same config and then compare the results. The actual rules are specified in configuration files by *`degradation.strategies`*. The strategies are specified as an ordered list, and all of the applicable rules are collected through all of the configurations (starting from the runtime configuration, through local ones, up to the global configuration). This yields a *list of rules* (each rule represented as key-value dictionary) ordered by the priority of their application. So for each pair of tested profiles, we iterate through this ordered list and find either the first that is applicable according to the set rules (by setting the *`degradation.apply`* key to value `first`) or all applicable rules (by setting the *`degradation.apply`* key to value `all`).

The example of configuration snippet that sets rules and strategies for one project can be as follows:

```
degradation:
  apply: first
  strategies:
    - type: mixed
      postprocessor: regression_analysis
      method: bmoe
    - cmd: mybin
      type: memory
      method: bmoe
    - method: aat
```

The following list of strategies will first try to apply the *Best Model Order Equality* method to either mixed profiles postprocessed by *Regression Analysis* or to memory profiles collected from command `mybin`. All of the other profiles will be checked using *Average Amount Threshold*. Note that applied methods can either be specified by their full name or using the short strings by taking the first letters of each word of the name of the method, so e.g. *BMOE* stands for *Best Model Order Equality*.

8.4 Create Your Own Degradation Checker

New performance change checkers can be registered within Perun in several steps. The checkers have just small requirements and have to *yield* the reports about degradation as a instances of `DegradationInfo` objects specified as follows:

class perun.check.**DegradationInfo**(*res*, *t*, *loc*, *fb*, *tt*, *ct='no'*, *cr=0*)

The returned results for performance check methods

Variables

- **result** (*PerformanceChange*) – result of the performance change, either can be optimization, degradation, no change, or certain type of unknown
- **type** (*str*) – string representing the type of the degradation, e.g. "order" degradation
- **location** (*str*) – location, where the degradation has happened
- **from_baseline** (*str*) – value or model representing the baseline, i.e. from which the new version was optimized or degraded
- **to_target** (*str*) – value or model representing the target, i.e. to which the new version was optimized or degraded
- **confidence_type** (*str*) – type of the confidence we have in the detected degradation, e.g. r^2
- **confidence_rate** (*int*) – value of the confidence we have in the detected degradation

You can register your new performance change checker as follows:

1. Run `perun utils create check my_degradation_checker` to generate a new modules in `perun/check` directory with the following structure. The command takes a predefined templates for new degradation checkers and creates `my_degradation_checker.py` according to the supplied command line arguments (see *Utility Commands* for more information about interface of `perun utils create` command):

```
/perun
|-- /check
    |-- __init__.py
    |-- average_amount_threshold.py
    |-- my_degradation_checker.py
```

2. Implement the `my_degradation_checker.py` file, including the module docstring with brief description of the change check with the following structure:

```
"""..."""

import perun.check as check


```

```
def my_degradation_checker(baseline_profile, target_profile):
    """..."""
    yield check.DegradationInfo("...")
```

3. Next, in the `__init__.py` module register the short string for your new method as follows:

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/degradation_init.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/degradation_init_new_check.py
@@ -3,8 +3,10 @@
     short_strings = {
         'aat': 'average_amount_threshold',
         'bmoe': 'best_model_order_equality',
+        'mdc': 'my_degradation_checker'
     }
     if strategy in short_strings.keys():
         return short_strings[strategy]
     else:
         return strategy
+
```

4. Preferably, verify that registering did not break anything in the Perun and if you are not using developer instalation, then reinstall Perun:

```
make test
make install
```

5. At this point you can start using your check using `perun check head`, `perun check all` or `perun check profiles`.

6. If you think your collector could help others, please, consider making Pull Request.

8.5 Degradation CLI

Command Line Interface contains group of two commands for running the checks in the current project—`perun check head` (for running the check for one minor version of the project; e.g. the current *head*) and `perun check all` for iterative application of the degradation check for all minor versions of the project. The first command is mostly meant to run as a hook after each new commit (obviously after successfull run o f‘‘perun run matrix‘‘ generating the new batch of profiles), while the latter is meant to be used for new projects, after crawling through the whole history of the project and collecting the profiles. Additionally `perun check profiles` can be used for an isolate comparison of two standalone profiles (either registered in index or as a standalone file).

8.5.1 perun check head

Checks for changes in performance between between specified minor version (or current *head*) and its predecessor minor versions.

The command iterates over all of the registered profiles of the specified *minor version* (*target*; e.g. the *head*), and tries to find the nearest predecessor minor version (*baseline*), where the profile with the same configuration as the tested target profile exists. When it finds such a pair, it runs the check according to the strategies set in the configuration (see *Configuring Degradation Detection* or *Perun Configuration files*).

By default the `hash` corresponds to the *head* of the current project.

```
perun check head [OPTIONS] <hash>
```

Arguments

<hash>
Optional argument

8.5.2 perun check all

Checks for changes in performance for the specified interval of version history.

The commands crawls through the whole history of project versions starting from the specified `<hash>` and for all of the registered profiles (corresponding to some *target* minor version) tries to find a suitable predecessor profile (corresponding to some *baseline* minor version) and runs the performance check according to the set of strategies set in the configuration (see *Configuring Degradation Detection* or *Perun Configuration files*).

```
perun check all [OPTIONS] <hash>
```

Arguments

<hash>
Optional argument

8.5.3 perun check profiles

Checks for changes in performance between two profiles.

The commands checks for the changes between two isolate profiles, that can be stored in pending profiles, registered in index, or be simply stored in filesystem. Then for the pair of profiles <baseline> and <target> the command runs the performance chekc according to the set of strategies set in the configuration (see *Configuring Degradation Detection* or *Perun Configuration files*).

<baseline> and <target> profiles will be looked up in the following steps:

1. If profile is in form `i@i` (i.e, an *index tag*), then *ith* record registered in the minor version <hash> index will be used.
2. If profile is in form `i@p` (i.e., an *pending tag*), then *ith* profile stored in `.perun/jobs` will be used.
3. Profile is looked-up within the minor version <hash> index for a match. In case the <profile> is registered there, it will be used.
4. Profile is looked-up within the `.perun/jobs` directory. In case there is a match, the found profile will be used.
5. Otherwise, the directory is walked for any match. Each found match is asked for confirmation by user.

```
perun check profiles [OPTIONS] <baseline> <target>
```

Options

-m, --minor <minor>
Will check the index of different minor version <hash> during the profile lookup.

Arguments

<baseline>
Required argument

<target>
Required argument

CHAPTER

NINE

PERUN CONFIGURATION FILES

Perun stores its configuration in Yaml format, either locally for each wrapped repository, or globally for the whole system (see *Configuration types*). Most of the configuration options is recursively looked up in the hierarchy, created by local and global configurations, until the option is found in the nearest configuration. Refer to *List of Supported Options* for description of options, such as formatting strings for status and log outputs, specification of job matrix (in more details described in *Job Matrix Format*) or information about wrapped repository.

In order to configure your local instance of Perun run the following:

```
perun config --edit
```

This will open the nearest local configuration in text editor (by default in `vim`) and lets you modify the options w.r.t. Yaml format.

9.1 Configuration types

Perun uses two types of configurations: **global** and **local**. The global configuration contains options shared by all of the Perun instances found on the host and the local configuration corresponds to concrete wrapped repositories (which can, obviously, be of different type, with different projects and different profiling information). Both global and local configurations have several options restricted only to their type (which is emphasized in the description of individual option). The rest of the options can then be looked up either recursively (i.e. first we check the nearest local perun instance, and traverse to higher instances until we find the searched option or eventually end up in the global configuration) or gathered from all of the configurations from the whole configuration hierarchy (ordered by the depth of the hierarchy, i.e. options found in global configuration will be on the bottom of the list). Options are specified by configuration sections, subsections and then concrete options delimited by `.`, e.g. `local.general.editor` corresponds to the `editor` option in the `general` section in `local` configuration.

The location of global configuration differs according to the host system. In UNIX systems, the **global** configuration can be found at:

```
$HOME/.config/perun
```

In Windows systems it is located in user storage:

```
%USERPROFILE%\AppData\Local\perun
```

9.2 List of Supported Options

vcs
: `[local-only]` Section, which contains options corresponding to the version control system that is wrapped

by instance of Perun. Specifies e.g. the type (in order to call corresponding auxiliary functions), the location in the filesystem or wrapper specific options (e.g. the lightweight custom `tagit` vcs constains additional options).

vcs.type
`[local-only]` Specifies the type of the wrapped version control system, in order to call corresponding auxiliary functions. Currently `git` is supported, with custom lightweight vcs `tagit` in development.

vcs.url
`[local-only]` Specifies path to the wrapped version control system, either as an absolute or a relative path that leads to the directory, where the root of the wrapped repository is (e.g. where `.git` is).

general
Section, which contains options and specifications potentially shared by more Perun instances. This section contains e.g. underlying text editor for editing, or paging strategy etc.

general.paging
Sets the paging for `perun log` and `perun status`. Paging can be currently set to the following four options: `always` (both `log` and `status` will be paged), `only-log` (only output of `log` will be paged), `only-status` (only output of `status` will be paged and `never`. By default `only-log` is used in the configuration. The behaviour of paging can be overwritten by option `--no-pager` (see *Command Line Interface*).

general.editor
`[recursive]` Sets user choice of text editor, that is e.g. used for manual text-editing of configuration files of Perun. Specified editor needs to be executable, has to take the filename as an argument and will be called as `general.editor config.yml`. By default `editor` is set to `vim`.

format
This section contains various formatting specifications e.g. formatting specifications for `perun log` and `perun status`.

format.status
`[recursive]` Specifies the formatting string for the output of the `perun status` command. The formatting string can contain raw delimiters and special tags, which are used to output concrete information about each profile, like e.g. command it corresponds to, type of the profile, time of creation, etc. Refer to *Customizing Statuses* for more information regarding the formatting strings for `perun status`.

E.g. the following formatting string:

```
| %type% | %cmd% | %workload% | %collector%  | (%time%) |
```

will yield the following status when running `perun status` (both for stored and pending profiles):

```
================================================================================*
  id |   type   |  cmd    | workload |  args   | collector  |         time         |
================================================================================*
 0@p | [mixed] | target | hello    |         | complexity | 2017-09-07 14:41:49 |
 1@p | [time ] | perun  |          | status | time       | 2017-10-19 12:30:29 |
 2@p | [time ] | perun  |          | --help | time       | 2017-10-19 12:30:31 |
================================================================================*
```

format.log
`[recursive]` Specifies the formatting string for the output of the short format of `perun log` command. The formatting string can contain raw characters (delimiters, etc.) and special tags, which are used to output information about concrete minor version (e.g. minor version description, number of assigned profiles, etc.). Refer to *Customizing Logs* for more information regarding the formatting strings for `perun log`.

E.g. the following formatting string:

```
'%id:6% (%stats%) %desc%'
```

will yield the following output when running `perun log --short`:

```
minor   (a|m|x|t profiles) info
53d35c  (2|0|2|0 profiles) Add deleted jobs directory
07f2b4  (1|0|1|0 profiles) Add necessary files for perun to work on this repo.
bd3dc3  ---no--profiles--- root
```

format.output_profile_template

`[recursive]` Specifies the format for automatic generation of profile files (e.g. when running `perun run job`, `perun run matrix`, `perun collect` or `perun postprocessby`. The formatting string consists either of raw characters or special tags, that output information according to the resulting profile. By default the following formatting string is set in the global configuration:

```
"%collector%-%cmd%-%args%-%workload%-%date%"
```

The supported tags are as follows:

`%collector%`:

Placeholder for the collection unit that collected the profiling data of the given profile. Refer to *Supported Collectors* for full list of supported collectors.

`%postprocessors%`:

Placeholder for list of postprocessors that were used on the given profile. The resulting string consists of postprocessor names joined by `-and-` string, i.e. for example this will output string `normalizer-and-filter`.

`%<unit>.<param>%`:

Placeholder for concrete value of `<param>` of one unit `<unit>` (either collector or postprocessor)

`%cmd%`:

Placeholder for the command that was profiled, i.e. some binary, script or command (refer to *cmds* or *Automating Runs* for more details).

`%args%`:

Placeholder for arguments that were supplied to the profiled command (refer to *args* or *Automating Runs* for more details).

`%workload%`:

Placeholder for workload that was supplied to the profiled command (refer to *workloads* or *Automating Runs* for more details).

`%type%`:

Placeholder for global type of the resources of the profile, i.e. *memory*, *time*, *mixed*, etc.

`%date%`:

Placeholder for the time and date that the profile was generated in form of `YEAR-MONTH-DAY-HOUR-MINUTES-SECONDS`.

`%origin%`:

Placeholder for the origin of the profile, i.e. the minor version identification for which the profiles was generated and the profiling data was collected.

`%counter%`:

Placeholder for increasing counter (counting from 0) for one run of perun. Note that this may rewrite existing profiles and is mostly meant to distinguish between profiles during one batch run of profile generation (e.g. when `perun run matrix` is executed).

cmds
`[local-only]` Refer to *cmds*.

args
`[local-only]` Refer to *args*.

workloads
`[local-only]` Refer to *workloads*

collectors
`[local-only]` Refer to *collectors*

postprocessors
`[local-only]` Refer to *postprocessors*

degradation
Section, which contains options and specifications potentially shared by more Perun instances. This section contains e.g. underlying text editor for editing, or paging strategy etc.

degradation.apply
`[recursive]`

degradation.strategies
`[gathered]` Specifies the rules for application of the performance degradation methods for profiles with corresponding profile configurations (e.g. with concrete profile type, specified collector, etc.). Refer to *Configuring Degradation Detection* for more details about application of strategies.

The following configuration will apply the *Best Model Order Equality* method for all of the *mixed* types of the profiles, which were postprocessed using the *Regression Analysis* and *Average Amount Threshold* otherwise.

```
degradation:
  strategies:
    - type: mixed
      postprocessor: regression_analysis
      method: bmoe
    - method: aat
```

9.3 Command Line Interface

We advise to manipulate with configurations using the `perun config --edit` command. In order to change the nearest local (resp. global) configuration run `perun config --local --edit` (resp. `perun config --shared --edit`).

9.3.1 perun config

Manages the stored local and shared configuration.

Perun supports two external configurations:

1. `local.yml`: the local configuration stored in `.perun` directory, containing the keys such as specification of wrapped repository or job matrix used for quick generation of profiles (run `perun run matrix --help` or refer to *Automating Runs* for information how to construct the job matrix).

2. `shared.yml`: the global configuration shared by all perun instances, containing shared keys, such as text editor, formatting string, etc.

The syntax of the `<key>` in most operations consists of section separated by dots, e.g. `vcs.type` specifies `type` key in `vcs` section. The lookup of the `<key>` can be performed in three modes, `--local`, `--shared` and `--nearest`, locating or setting the `<key>` in local, shared or nearest configuration respectively (e.g. when one is trying to get some key, there may be nested perun instances that do not contain the given key). By default, perun operates in the nearest config mode.

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options.

E.g. using the following one can retrieve the type of the nearest perun instance wrapper:

```
$ perun config get vsc.type
vcs.type: git
```

```
perun config [OPTIONS] COMMAND [ARGS]...
```

Options

-l, --local
Will lookup or set in the local config i.e. `.perun/local.yml`.

-h, --shared
Will lookup or set in the shared config i.e. `shared.yml`.

-n, --nearest
Will recursively discover the nearest suitable config. The lookup strategy can differ for `set` and `get`/`edit`.

Commands

edit
Edits the configuration file in the external...

get
Looks up the given `<key>` within the...

set
Sets the value of the `<key>` to the given...

9.3.2 perun config get

Looks up the given `<key>` within the configuration hierarchy and returns the stored value.

The syntax of the `<key>` consists of section separated by dots, e.g. `vcs.type` specifies `type` key in `vcs` section. The lookup of the `<key>` can be performed in three modes, `--local`, `--shared` and `--nearest`, locating the `<key>` in local, shared or nearest configuration respectively (e.g. when one is trying to get some key, there may be nested perun instances that do not contain the given key). By default, perun operates in the nearest config mode.

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options.

E.g. using the following can retrieve the type of the nearest perun wrapper:

```
$ perun config get vsc.type
vcs.type: git

$ perun config --shared get general.editor
general.editor: vim
```

```
perun config get [OPTIONS] <key>
```

Arguments

<key>
Required argument

9.3.3 perun config set

Sets the value of the <key> to the given <value> in the target configuration file.

The syntax of the <key> corresponds of section separated by dots, e.g. vcs.type specifies type key in vcs section. Perun sets the <key> in three modes, --local, --shared and --nearest, which sets the <key> in local, shared or nearest configuration respectively (e.g. when one is trying to get some key, there may be nested perun instances that do not contain the given key). By default, perun will operate in the nearest config mode.

The <value> is arbitrary depending on the key.

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options and their values.

E.g. using the following can set the log format for nearest perun instance wrapper:

```
$ perun config set format.shortlog "| %origin% | %collector% |"
format.shortlog: | %origin% | %collector% |
```

```
perun config set [OPTIONS] <key> <value>
```

Arguments

<key>
Required argument

<value>
Required argument

9.3.4 perun config edit

Edits the configuration file in the external editor.

The used editor is specified by the *general.editor* option, specified in the nearest perun configuration..

Refer to *Perun Configuration files* for full description of configurations and *Configuration types* for full list of configuration options.

```
perun config edit [OPTIONS]
```

CHAPTER

TEN

CUSTOMIZE LOGS AND STATUSES

`log` and `status` commands print information about wrapped repository annotated by performance profiles. `perun log` command lists the minor versions history for a major version (currently the checked out), along with the information about registered profiles, such as e.g. the minor version description, authors, statistics of profiles, etc. `perun status` commands shows the overview of given minor version of current major head and lists profiles associated to profiles and in pending directory (i.e. the `.perun/jobs` directory. List of profiles contains the types of profiles, numbers, configurations of profiling run, etc.

The format of outputs of both `log` and `status` can be customized by setting the formatting strings c.f. *Customizing Logs* and *Customizing Statuses*. Moreover, outputs are paged (currently using the `less -R` command) by default. To turn off the paging, run the `perun` with `--no-pager` option (see *Command Line Interface*) or set *`general.paging`*.

10.1 Customizing Statuses

The output of `perun status` is defined w.r.t. formatting string specified in configuration in *`format.status`* key (looked up recursively in the nearest local configuration, or in global configuration). The formatting string consists of raw delimiters and special tags, which serves as templates to output specific informations about concrete profiles, such as the profiling configuration, type of profile, creating timestamps, etc.

E.g. the following formatting string:

```
| %type% | %cmd% | %workload% | %collector%  | (%time%) |
```

will yield the following status when running `perun status` (both for stored and pending profiles):

```
=================================================================================*
  id |   type   |  cmd   | workload |  args   | collector  |         time         |
=================================================================================*
 0@p | [mixed] | target | hello    |         | complexity | 2017-09-07 14:41:49 |
 1@p | [time ] | perun  |          | status | time       | 2017-10-19 12:30:29 |
 2@p | [time ] | perun  |          | --help | time       | 2017-10-19 12:30:31 |
=================================================================================*
```

The first column of the `perun status` output, `id`, has a fixed position and defines a tag for the given, which can be used in `add`, `rm`, `show` and `postprocessby` commands as a quick wildcard for concrete profiles, e.g. `perun add 0@p` would register the first profile stored in the pending `.perun/jobs` directory to the index of current head. Tags are always in form of `i@p` (for pending profiles) and `i@i` for profiles registered in index, where `i` stands for position in the corresponding storage, index from zero.

The specification of the formatting string can contain the following special tags:

`%type%`: Lists the most generic type of the profile according to the collected resources serving as quick tagging of similar profiles. Currently Perun supports *memory*, *time*, *mixed*.

%cmd%: Lists the command for which the data was collected, this e.g. corresponds to the binary or script that was executed and profiled using collector/profiler. Refer to *Overview of Jobs* for more information about profiling jobs and commands.

%args%: Lists the arguments (or parameters) which were passed to the profiled command. Refer to *Overview of Jobs* for more information about profiling jobs and command arguments.

%workload%: List input workload which was passed to the profiled command, i.e. some inputs of the profiled program, script or binary. Refer to *Overview of Jobs* for more information about profiling jobs and command workloads.

%collector%: Lists the collector which was used to obtain the given profile. Refer to *Collectors Overview* for list of supported collectors and more information about collection of profiles.

%time%: Timestamp when the profile was last modified in format *YEAR-MONTH-DAY HOURS:MINUTES:SECONDS*.

%id%: Original identification of the profile. This corresponds to the name of the generated profile and the original path.

10.2 Customizing Logs

The output of `perun log --short` is defined w.r.t. formatting string specified in configuration in *`format.log`* key (looked up recursively in the nearest local configuration, or in global configuration). The formatting string can contain both raw characters (such as delimiters, etc.) and special tags, which serves as templates to output information for concrete minor version such as minor version description, number of assigned profiles, etc.

E.g. the following formatting string:

```
'%checksum:6% (%stats%) %desc%'
```

will yield the following output when running `perun log --short`:

```
minor   (a|m|x|t profiles) info
53d35c  (2|0|2|0 profiles) Add deleted jobs directory
07f2b4  (1|0|1|0 profiles) Add necessary files for perun to work on this repo.
bd3dc3  ---no--profiles--- root
```

The specification of the formatting string can contain the following special tags:

%checksum:num%: Identification of the minor version (should be hash preferably). If we take `git` as an example `checksum` will correspond to the SHA of one commit.

%stats%: Lists short summary of overall number of profiles (`a`) and number of memory (`m`), mixed (`x`) and time (`t`) profiles assinged to given minor version.

%desc:num%: Lists short description of the minor version, limiting to the first sentence of the description. If we take `git` as an example this will correspond to the short commit message.

%date:num%: Lists the date the minor version was commited (in the wrapped vcs).

%author:num%: Lists the author of the minor version (not commiter).

%email:num%: Lists the email of the author of the minor version.

%parents:num%: Lists the parents of the given minor version. Note that one minor version can have potentially several parents, e.g. in git, when the merge of two commits happens.

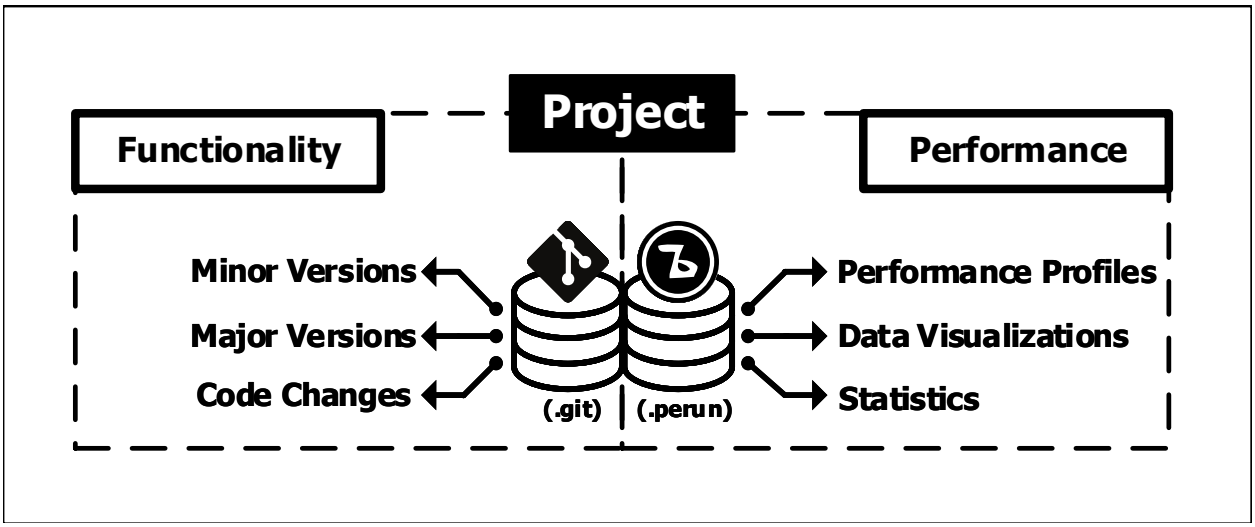
Specifying `num` in the selected tags will shorten the displayed identification to `num` characters only. In case the specified `num` is smaller then the length of the attribute name, then the shortening will be limited to the lenght of the attribute name.

CHAPTER

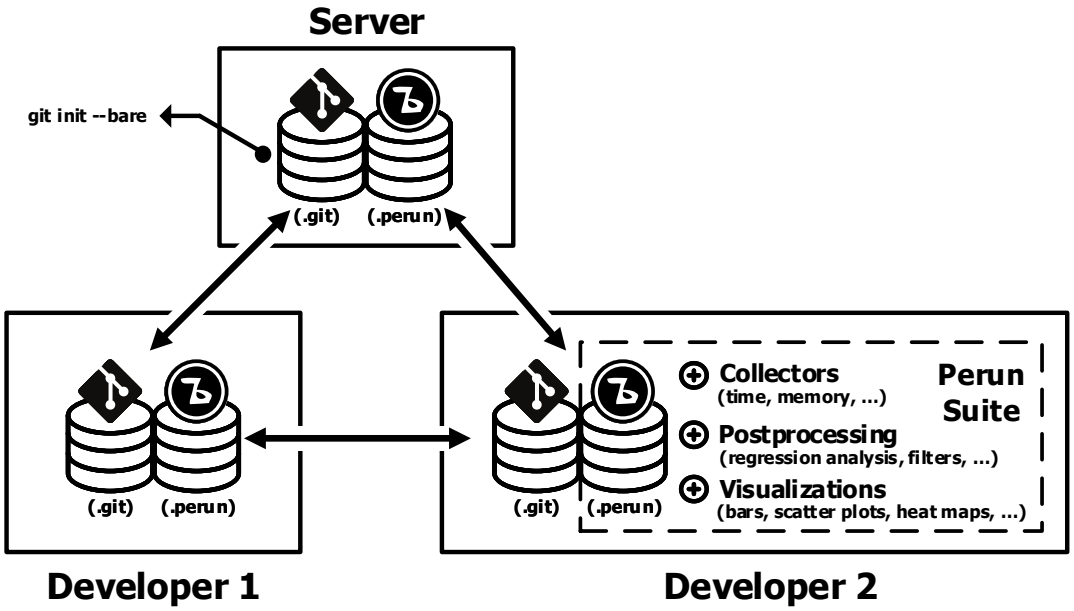
ELEVEN

PERUN INTERNALS

Conceptually one Perun instances serves as a wrapper around the existing version control system (e.g. some repository). Perun takes specializes on storing the performance profiles and manages the link between minor versions and their corresponding profiles. Currently as a target vcs we support only `git`, with a custom lightweigth vcs being in development (called `tagit`). The architecture of Perun contains an interface that can be used to register support for new version control system as described in *Creating Support for Custom VCS*. Internal structure of one instance of Perun is inspired by git: performance profiles are similarly stored as objects compressed by zlib method and identified by hashes. *Perun Storage* describes the internal model of Perun more briefly.



The diagram above highlights the responsibilities and storage of individual systems. Version control systems manage the functionality of the project—its versions and precise code changes—but lack proper support for managing performance. On the other hand, performance versioning systems manages the performance of project—its individual performance profiles, data visualizations of various statistics—but lack the precise functionality changes. This means that vcs stores the actual code chungs and version references and pvs stores the actual profiling data.



This diagram shows one of the proper usages of Perun's tool suite. Each developer keeps his own instance of both versioning and performance systems. In this mode one can share both the code changes and performance measurement through the wider range of developers.

11.1 Version Control Systems

Version Control System manages the history of functionality of one project, i.e. stores the changes between different versions (or snapshots) of project. Each code change usually requires corresponding the performance profiles in order to detect potential performance degradation early in the development. The following subsection *Version Control System API* describes the layer which serves as an interface in Perun which supplies the necessary information between the version control and performance versioning systems.

11.1.1 Version Control System API

`perun.vcs.`**`init`**(*vcs_type*, *vcs_path*, *vcs_init_params*)

Calls the implementation of initialization of wrapped underlying version control system.

The initialization should take care of both reinitialization of existing version control system instances and newly created instances. Init is called during the `perun init` command from command line interface.

Parameters

- **`vcs_type`** (*`str`*) – type of the underlying wrapped version control system
- **`vcs_path`** (*`str`*) – destination path of the initialized wrapped vcs
- **`vcs_init_params`** (*`dict`*) – dictionary of keyword arguments passed to initialization method of the underlying vcs module

Returns true if the underlying vcs was successfully initialized

perun.vcs.**walk_minor_versions**(*vcs_type*, *vcs_path*, *head_minor_version*)
Generator of minor versions for the given major version, which yields the `MinorVersion` named tuples containing the following information: `date`, `author`, `email`, `checksum` (i.e. the hash representation of the minor version), `commit_description` and `commit_parents` (i.e. other minor versions).

Minor versions are walked through this function during the `perun log` command.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs
- **head_minor_version** (*str*) – the root minor versions which is the root of the walk.

Returns iterable stream of minor version representation

perun.vcs.**walk_major_versions**(*vcs_type*, *vcs_path*)
Generator of major versions for the current wrapped repository.

This function is currently unused, but will be needed in the future.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs

Returns interable stream of major version representation

perun.vcs.**get_minor_head**(*vcs_type*, *vcs_path*)
Returns the string representation of head of current major version, i.e. for git this returns the massaged HEAD reference.

This function is called mainly during the outputs of `perun log` and `perun status` but also during the automatic generation of profiles (either by `perun run` or `perun collect`), where the retrieved identification is used as *origin*.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs

Returns unique string representation of current head (usually in SHA)

Raises **ValueError** – if the head cannot be retrieved from the current context

perun.vcs.**get_head_major_version**(*vcs_type*, *vcs_path*)
Returns the string representation of current major version of the wrapped repository.

Major version is displayed during the `perun status` output, which shows the current working major version of the project.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs

Returns string representation of the major version

perun.vcs.**get_minor_version_info**(*vcs_type*, *vcs_path*, *minor_version*)
Yields the specification of concrete minor version in form of the `MinorVersion` named tuples containing the following information: `date`, `author`, `email`, `checksum` (i.e. the hash representation of the minor version), `commit_description` and `commit_parents` (i.e. other minor versions).

This function is a non-generator alternative of *perun.vcs.walk_minor_versions()* and is used during the perun status output to display the specifics of minor version.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs
- **minor_version** (*str*) – the specification of minor version (in form of sha e.g.) for which we are retrieving the details

Returns minor version named tuple

perun.vcs.**check_minor_version_validity**(*vcs_type*, *vcs_path*, *minor_version*)

Checks whether the given minor version specification corresponds to the wrapped version control system, and is not in wrong format.

Minor version validity is mostly checked during the lookup of the minor versions from the command line interface.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs
- **minor_version** (*str*) – the specification of minor version (in form of sha e.g.) for which we are checking the validity

Raises **VersionControlSystemException** – when the given minor version is invalid in the context of the wrapped version control system.

perun.vcs.**massage_parameter**(*vcs_type*, *vcs_path*, *parameter*, *parameter_type=None*)

Conversion function for massaging (or unifying different representations of objects) the parameters for version control systems.

Massaging is mainly executed during from the command line interface, when one can e.g. use the references (like HEAD) to specify concrete minor versions. Massing then unifies e.g. the references or proper hash representations, to just one representation for internal processing.

Parameters

- **vcs_type** (*str*) – type of the underlying wrapped version control system
- **vcs_path** (*str*) – source path of the wrapped vcs
- **parameter** (*str*) – vcs parameter (e.g. revision, minor or major version) which will be massaged, i.e. transformed to unified representation
- **parameter_type** (*str*) – more detailed type of the parameter

Returns string representation of parameter

11.1.2 Creating Support for Custom VCS

You can register support for your own version control system as follows:

1. Create a new module in perun/vcs directory implementing functions from *Version Control System API*.
2. Finally register your newly created vcs wrapper in get_supported_module_names() located in perun.utils.__init__.py:

```
--- /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/f/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪collectors.py
@@ -6,7 +6,7 @@
         ))
     return {
         'vcs': ['git'],
-        'collect': ['complexity', 'memory', 'time'],
+        'collect': ['complexity', 'memory', 'time', 'mycollector'],
         'postprocess': ['filter', 'normalizer', 'regression_analysis'],
         'view': ['alloclist', 'bars', 'flamegraph', 'flow', 'heapmap', 'raw
↪', 'scatter']
     }[package]
```

3. Optionally implement batch of automatic test cases using (preferably based on pytest) in `tests` directory. Verify that registering did not break anything in the Perun, your wrapper is correct and optionally reinstall Perun:

```
make test
make install
```

4. If you think your wrapper could help others, please, consider making Pull Request.

11.2 Perun Storage

The current internal representation of Perun storage is based on git internals and is meant for easy distribution, flexibility and easier managing. The possible extension of Perun to different versions of storages is currently under consideration. Internal objects and files for one local instance of Perun are stored in the filesystem in the `.perun` directory consisting of the following infrastructure:

```
.perun/
    |-- /jobs
    |-- /objects
    |-- local.yml
```

`.perun/jobs`: Contains pending jobs, i.e. those that were generated by collectors, postprocessed by some postprocessors, or automatically generated by `perun run` commands, but are not yet assigned to concrete minor versions. These profiles contains the tag *`origin`* that maps the profile to concrete minor version, i.e. the parent of the profile. This key serves as a prevention of assigning profiles to incorrect minor versions.

```
.perun/jobs
    |-- /baseline.perf
    |-- /sll-comparison.perf
    |-- /skip-lists-medium-height.perf
    |-- /skip-lists-unlimited-height.perf
```

`.perun/objects`: Corresponds to main storage of Perun and contains object primitives. Every object of Perun is represented by unique identifier (mostly by sha representation) and corresponds either to an object blob (containing compressed profile) or to an index of a corresponding minor version, which lists assigned profiles for the given minor version.

```
.perun/objects
    |-- /07
        |-- f2b4bfa06f6b1be5713f2bbae7740838456758
        |-- 99dc4c5891947bdf7e26341231ca533432a1f1
```

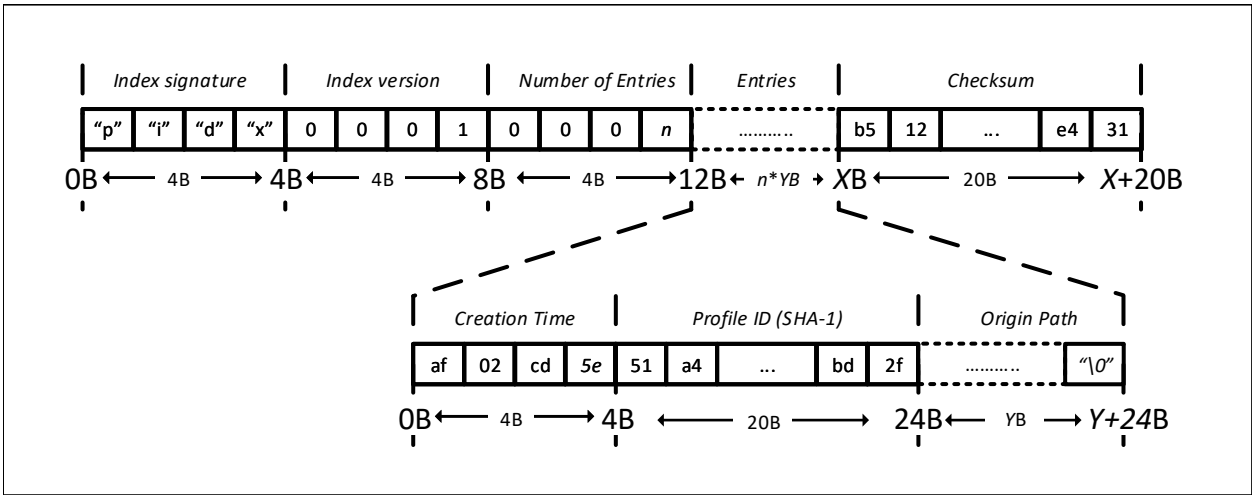
```
    |-- /3d
        |-- 3859b46db4eea5866a0b2b28997fac25a95430
    |-- /ff
        |-- d35c8962d8d2019d7762a7bc6980c1d0f2fcd7
        |-- d88aabca6e5427c78ea647e955ffa00d1cd615
```

Each object from `.perun/objects` is represented by hash value, where the first two characters are used to specify directory and the rest of the hash value a file name, where the index or compressed file is stored.

`local.yml`: Contains local configuration, e.g. the specification of wrapped repository, job matrixes or formatting strings corresponding to concrete VCS. See *Perun Configuration files* for more information about configuration of Perun.

11.2.1 Perun Index Specification

Each minor version of vcs, which has any profile assigned, has corresponding index file in the `.perun/object` according to its identification. The index file itself is stored in binary format with the following specification.



***Index signature* [4B]:** Signature are the first bytes of the index containing ascii string *pidx*, which serves as an quick identification of minor version index.

***Index version* [4B]:** Specification of version of conding of the index. Versioning is introduced for potential future backward compatibility with possible different specifications of index.

***Number of Entries* [4B]:** Integer count of the number of entries found in the index. Each entry of the index is of variable length and lists the profiles with mapping to their corresponding objects.

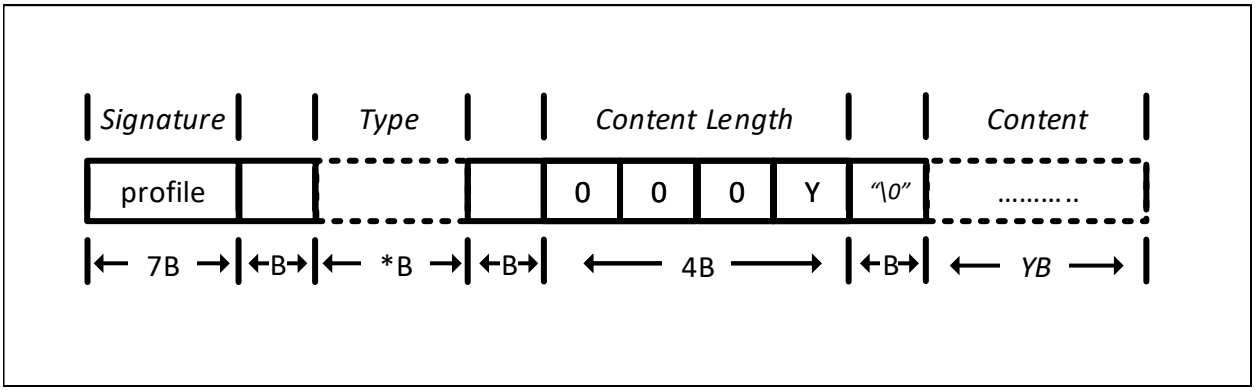
***Entries* [*variable length*]:** One entry of the index corresponds to one assigned profile. Each entry is of variable lenght and contains the identification of the original profile file, together with timestamp of creation and the identification of the compressed object, that contains the actual profiling data. Each entry can be broken into following parts:

- *Creation time* [4B]: creation time of the profile represented as 4B timestamp.
- *Profile ID* [20B]: unique identification of the profile, i.e. specification of the concrete compressed object located in the `.perun/objects`. Profile ID is always in form of SHA-1 hash, which is obtained from the contents.
- *Origin Path* [*variable length*: Original path to the profile represented as ascii string of variable length terminated by null byte.

***Checksum* [20B]:** Checksum of the whole index, which serves for error detection.

11.2.2 Perun Object Specification

Each non-index object consist of short header ended with zero byte, consisting of header signature string, type of the profile and lenght of the content, and raw content of the performance profile w.r.t. *Specification of Profile Format*. First we compute the checksum for these data, which serves as an identification in the minor version indexes and in `.perun/objects` directory. Finally, the object is compressed using zlib method and stored in the `.perun/objects` compressed.



***Signature* [7B]:** Signature is a 7B prefix containing ascii string "profile". Serves for quick identification of profile.

***Type* [*variable length*]:** Ascii specification of the profile type. This serves for quick and easy parsing of profiles.

***Content Lenght* [4B]:** Integer count of the non-header data followed after the zero byte in bytes.

***Content* [*variable length*]:** Contents of the performance profile w.r.t. *Specification of Profile Format*.

11.2.3 The Lifetime of profile: Internals

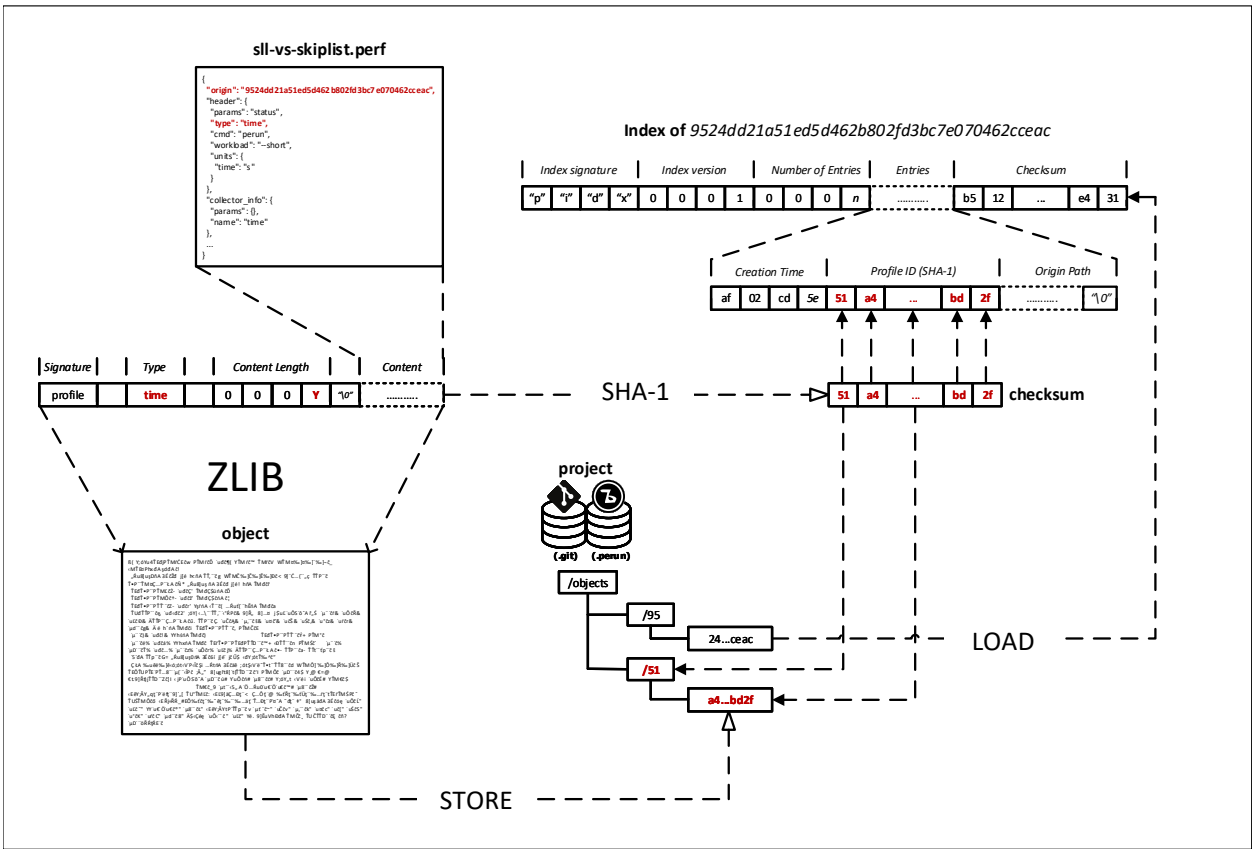
The following subsections describes in more detail the basics of profile manipulations, namely registering, removing and lookuping up profiles.

Registering new profile

Given a profile, w.r.t. *Specification of Profile Format*, called `sll-vs-skiplist.perf`, registering this profile in `HEAD` minor version index, the following steps are executed:

1. `sll-vs-skiplist.perf` is loaded and parsed into JSON. Profile is verified whether it is in format specified by *Specification of Profile Format*.
2. *`origin`* key is compared with the massaged `HEAD` minor version. In case it differes, an error is raised and adding the profiles is canceled, as we are trying to register performance profile corresponding to other point of history. Otherwise the *`origin`* is removed from the profile and will not be stored in persistent storage.
3. We construct the header for the profile consisting of `profile` prefix, the type of the specified by `type` and length of the unpacked JSON representation of profile, joined by spaces and ended by null byte.
4. JSON contents of performance profile are appended to the header resulting into one `object`.
5. An SHA-1 hash `checksum` is computed out of the `object`. The hash serves both as a check that the profile was not damaged during next usage, as well as identification in the filesystem.

6. The `object` is compressed using *zlib* compression method and stored in the `.perun/objects` directory. First two characters of `checksum` specifies the target directory and the rest specifies the resulting filename.
7. An index corresponding to the `HEAD` minor version is opened (if it does not exist, it is newly created first). Minor version index is also represented by its hash, where first two characters of hash is used as directory and the rest as filename.
8. An entry for `sll-vs-skiplist.perf` with given modification time is registered within the index pointing to the `checksum` object with compressed data. The number of registered profiles in index is increased.
9. Unless it is specified otherwise, the `sll-vs-skiplist.perf` is removed from filesystem.



Removing profile from index

Given a profile filename `sll-vs-skiplist.perf`, removing it from the `HEAD` minor version index, requires the following steps to be executed:

1. An index corresponding to the `HEAD` minor version is opened. Minor version index is represented by its hash, where first two characters of hash is used as directory and the rest as filename. If the index does not exist, removing ends with an error.
2. An entry for `sll-vs-skiplist.perf` is looked up within within the index. If it is not found, the removing ends with an error. Other wise, the entry is removed from the index and the number of registered profiles in index is decreased.
3. The original compresed object, which was stored in the entry is kept in the `.perun/objects` directory.

Looking up profile

Profiles are looked-up during the `perun show`, `perun add`, `perun postprocessby` or `perun rm` and can be found in several places, namely the filesystem, pending storage or registered in index. Priorities during the lookup are usually as follows:

1. If the specification of profile is in form of `i@i` or `i@p` (i.e. the *index* and *pending* tags respectively), then `i` th profile registered in index or stored in pending jobs directory (`.perun/jobs`) is used.
2. Index of corresponding minor version is searched.
3. Absolute path in filesystem is checked.
4. `.perun/jobs` directory is searched for match, i.e. one can specify just partial name of the profile during the lookup.
5. Otherwise the whole scope of filesystem is walked. Each successful match asks user for confirmation until the profile is found.

Refer to *Command Line Interface* for precise specification of lookups during individual commands.

CHAPTER

TWELVE

CHANGELOG

12.1 HEAD

To be included in next release

- add SystemTap based complexity collector
- add `perun utils create` command (see *Utility Commands* for more details) for creating new modules according to stored templates
- fix issue with getting config hierarchy, when outside of any perun scope

12.2 0.12.1 (2018-03-08)

commit: 96ef4443244568260e5dd25fa4cde5230eba8a36

Update project readme

- update the project readme
- add compiled documentation

12.3 0.12 (2018-03-05)

commit: 7ac008e0a7be32d5ddfceb3cbe7042036323f82d

Add basic testing of performance changes between profiles

- add command for checking performance changes between two isolate profiles
- add command for checking performance changes in given minor version
- add command for checking performance changes within the project history
- add two basic methods of checking performance changes
- add two options to config (see `degradation.strategies` and `degradation.apply`) to customize performance checking
- add caching to recursive config lookup
- add recursive gathering of options from config
- fix nondeterministic tests
- define structure for representing the result of performance change

- add basic implementation of performance change detectors

12.4 0.11.1 (2018-02-28)

commit: 8a6b1ac90c4cfca6f11546d0d3c4aa4fbe2000c3

Enhance the regression model suite

- fix issues when reading configuration with error
- enhance the regression model suite by improving quadratic and constant models
- rename the tags to different format (%tag%)
- add support for shortlog formatting string
- fix issue with postprocessing information being lost
- add options for changing filename template
- remodel automatic generation of profile names (now templatable; see *format.output_profile_template*)
- add runtime config
- break config command to three (get, set, edit)
- rename some configuration options
- fix issue with missing header parts in profiles
- fix issue with incorrect parameter
- add global.paging option (see *general.paging*)
- improve bokeh outputs (with click policy, and better lines)
- other various fixes

12.5 0.11 (2017-11-27)

Adding proper documentation

commit: a2ad710aafa171dfc6974c7121b572ee3ea2033b

- add HTML and latex documentation
- refactor the documentation of publicly visible modules
- add additional figures and examples of outputs and profiles
- switch order of initialization of Perun instances and vcs
- break vcs-params to vcs-flags and vcs-param
- fix the issue with missing index
- enhance the performance of Perun (guarding, rewriting to table lookup, or lazy inits)
- add loading of yaml parameters from CLI

12.6 0.10.1 (2017-10-24)

Remodeling of the regression analysis interface

commit: 14ce41c28d4d847ed2c74eac6a2dbfe7644cfd93

- refactor the interface of regression analysis
- update the regression analysis error computation
- add new parameters for plotting models
- reduce number of specific computation functions
- update the architecture (namely the interface)
- update the documentation of regression analysis and parameters for cli
- update the regressions analysis error computation
- add constant model
- add paging for perun log and status
- rename converters and transformations modules

12.7 0.10 (2017-10-10)

Add Scatter plot visualization module

commit: f0d9785639e5c03a994eb439d54206722a455da3

- add scatter plot as new visualisation module (basic version with some temporary workarounds)
- fix bisection method not producing model for some intervals
- add examples of scatter plot graphs

12.8 0.9.2 (2017-09-28)

Extend the regression analysis module

commit: 12c06251193701356685e8163a7ef8ce8b7d9f2a

- add transformation of models to plotable data points
- add helper functions for plotting models
- add support of regression analysis extensions

12.9 0.9.1 (2017-09-24)

Extend the query module

commit: bf8ff341cfa942b82093850c63655b79674ea615

- add proper testing to query module
- polish the messy conftest.py

- add support generators and fixtures for query profiles
- extend the profile query module with key values and models queries

12.10 0.9 (2017-08-31)

Add regression analysis postprocessing module

commit: 2b3d0d637699ae35b36672df3ce4c14fa0fed701

- add regression analysis postprocessor module
- add example resulting profiles

12.11 0.8.3 (2017-08-31)

commit: e47f5588e834fd70042bb18ea53a7d76f75cc8b7

Update and fix complexity collector

- fix several minor issues with complexity collector
- polish the standard of the generated profile
- add proper testinr for cli
- refactor according to the pylint
- fix bug where vector would not be cleared after printing to file
- remove code duplication in loop specification
- fix different sampling data structure for job and complexity cli
- fix some minor details with cli usage and info output

12.12 0.8.2 (2017-07-31)

Update the command line interface of complexity collector

commit: 1451ae054e77e81bf0aa4930639bf323c09c510e

- add new options to complexity collector interface
- add thorough documentation
- refactor the implementation

12.13 0.8.1 (2017-07-30)

Update the performance of command line interface

commit: 1fef373e8899b3ff0b0525ec99da91ba7a67fac0

- add on demand import of big libraries
- optimize the memory collector by minimizing subprocess calls

- fix issue with regex in memory collector
- add caching of memory collector syscalls
- extend cli of add and remove to support multiple args
- extend the massaging of parameters for cli
- remodel the config command
- add support for tags in command line
- enhance the status output of the profile list
- enhance the default formatting of config
- add thorough validity checking of bars/flow params

12.14 0.8 (2017-07-03)

Add flame graph visualization

commit: 56a29c807f2d7ad34b7af6002e5ebf90c717e8d7

- add flame graph visualization module

12.15 0.7.2 (2017-07-03)

Refactor flow graph to a more generic form

commit: eb33811236575599fc9aa82ce417c492be22d79b

- refactor flow to more generic format
- work with flattened pandas.DataFrame format
- use set of generators and queries for manipulation with profiles
- make the cli API generic
- polish the visual apeal of flow graphs
- simplify output to bokeh.charts.Area
- add basic testing of bokeh flow graphs
- fix the issue with additional layer in memory profs

12.16 0.7.1 (2017-06-30)

Refactor bar graph to a more generic form

commit: 5942e0b1aa8cc09ce0e22b030c3ec17dfdce0556

- refactor bars to more generic format
- work with flattened pandas.DataFrame format
- make the cli API generic
- polish the visual apeal of bars graph

- add unique colour palette to bokeh graphs
- fix minor issue with matrix in config
- add massaging of params for show and postprocess

12.17 0.7 (2017-06-26)

Add bar graph visualization

commit: a0f1a4921ecf9ef8f5b7c14ba42442fc589581ed

- integrate bar graph visualization

12.18 0.6 (2017-06-26)

Add Flow graph visualization

commit: 5683141b2e622af871eabc1c7259654151177256

- integrate flow graph visualization

12.19 0.5.1 (2016-06-22)

Fix issues in memory collector

commit: 28560e8d47cb2b1e2087d7072c44584563f78870

- extend the CLI for memory collect
- annotate phases of memory collect with basic informations
- add checks for presence of debugging symbols
- fix in various things in memory collector
- extend the testing of memory collector

12.20 0.5 (2016-06-21)

Add Heap map visualization

commit: 6ac6e43080f0a9b0c856636ed5ae12ee25a3d4df

- integrate Heap map visualization
- add thorough testing of heap and heat map
- refactor profile converting
- refactor duplicate blobs of code
- add animation feature
- add origin to profile so it can be compared before adding profile
- add more smart lookup of the profile for add

- add choices for collector/vcs/postprocessor parameters in cli
- simplify adding parameters to collectors/postprocessors
- add support for formatting strings for profile list
- refactor log and status function
- add basic testing for the command line interface
- switch interactive configuration to using editor
- implement wrappers for collect and postprocessby
- rename 'bin' keyword to 'cmd' in stored profiles
- add basic testing of the collectors and commands

12.21 0.4.2 (2017-05-31)

Collective fixes mostly for Memory collector

commit: 4d94299bc196292284995aabdce0c702e76b33ca

- fix a collector issue with zero value addresses
- add checking validity of the looked up minor version
- fix issue with incorrect parameter of the NotPerunRepositoryException
- raise exception when the profile is in incorrect json syntax
- catch error when minor head could not be found
- add exception for errors in wrapped VCS
- add exception for incorrect profile format
- raise NotPerunRepository, when Perun is not located on path
- fix message when git was reinitialized
- catch exceptions for init

12.22 0.4.1 (2017-05-15)

Collective fixes mosty for Complexity collector

commit: 13bebd88613fce58458d50207aea01ee7f672f86

- fixed size data container growth if functions were sampled
- enhance the perun status with info about untracked profiles
- add colours to printing of profile list (red for untracked)
- add output of untracked profiles to perun status
- fix issue with postprocessor parameter rewritten by local variable

12.23 0.4 (2017-03-17)

Add Complexity collector

commit: 323228f95050e52041b47af899eaea6e90eb0605

- add complexity collector module

12.24 0.3 (2017-03-14)

Adding Memory Collector

commit: 558ae1eee3acd370c519ac39e774d7fe05d23e35

- add memory collector module
- fix the issue with detached head state and perun status
- add simple, but interactive, initialization of the local config

12.25 0.2 (2017-03-07)

Add basic job units

commit: 7994b5618eb27684da57ce0941f4f58604ac29ea

- add the normalizer postprocessor
- add the time collector
- refactor the git module to use the python package
- add loadinng of config from local yml
- refactor construction of job matrix
- remove cmd from job tuple and rename params to args
- break perun run to run matrix (from config) and run job (from stdout)
- fix issue of assuming different structure of profile
- add functionality of creating and storing profiles
- add generation of the profile name for given job
- add storing of the profile at given path
- add generation of profile out of collected data
- update the params between the phases
- polish the perun –short header
- various minor tweaks for outputs
- change init-vcs-* options to just vcs-*
- fix an issue with incorrectly outputed comma if no profile type was present
- fix an issue with loading profile having two modes (compressed and uncompressed)
- implement base logic for calling collectors and postprocessors

- enhance output of profile numbers in perun log and status with colours and types
- add header for short info
- add colours to the header
- add base implementation of perun show
- fix loading of compressed file
- polish output of perun log and status by adding indent, colours and padding
- fix an issue with adding non-existent profile
- fix multiple adding of the same entry
- fix an issue when the added entry should go to end of index

12.26 0.1 (2017-02-22)

First partially working implementation

commit: 4dd5ee3c638570489d60c50ca41b519029da9007

- add short printing of minor version info (–short-minors | -s option)
- fix reverse output of log (oldest was displayed first)
- implement simplistic perun log outputing minor version history and profile numbers
- fix an incorrect warning about already tracked profiles
- add removal of the entry from the index
- add registering of files to the minor version index
- refactor according to pylint
- add base implementation of perun log
- add base implementation of perun status
- add base implementation of perun add
- add base implementation of perun rm
- add base implementation of perun init
- add base implementation of perun config
- add base commandline interface through click

12.27 0.0 (2016-12-10)

Initial minimalistic repository

commit: 2a6d1e65e5f3871e091d395789b9fd44450ef9e4

- empty root

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

F

G

H

I

L

M

O

P

R

S

T

U

V

W