Perun Documentation

Release 0.18.1

Tomas Fiedor, Jiri Pavela, et al.

Feb 13, 2020

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?

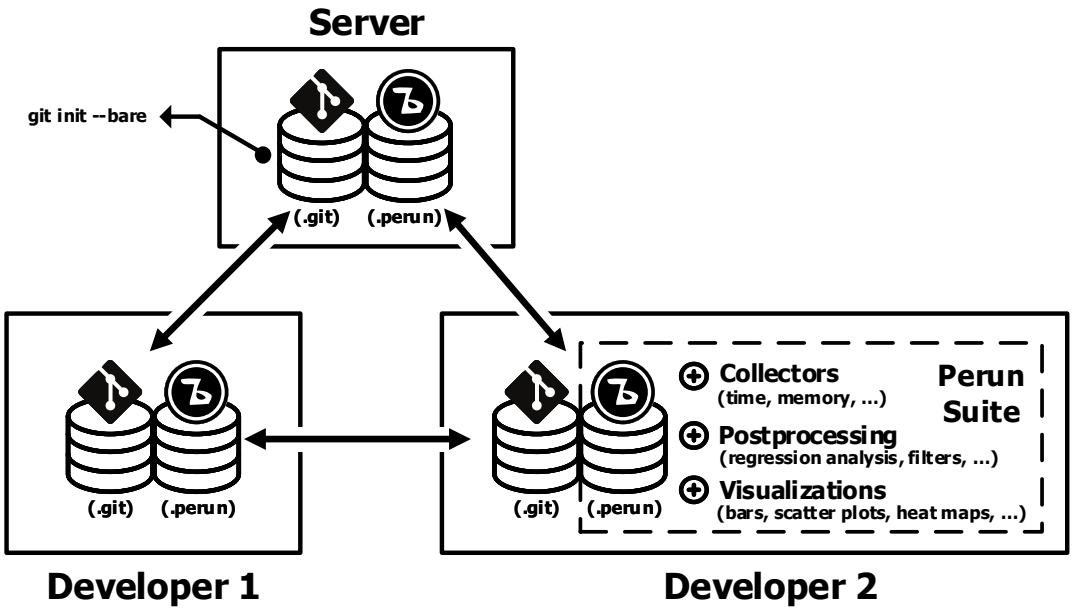
There exists solutions for managing changes of ones project—Version Control Systems (VCS)—but precise managing of performance is harder. This calls for solutions tuned to support performance management—Performance Versioning Systems.

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.

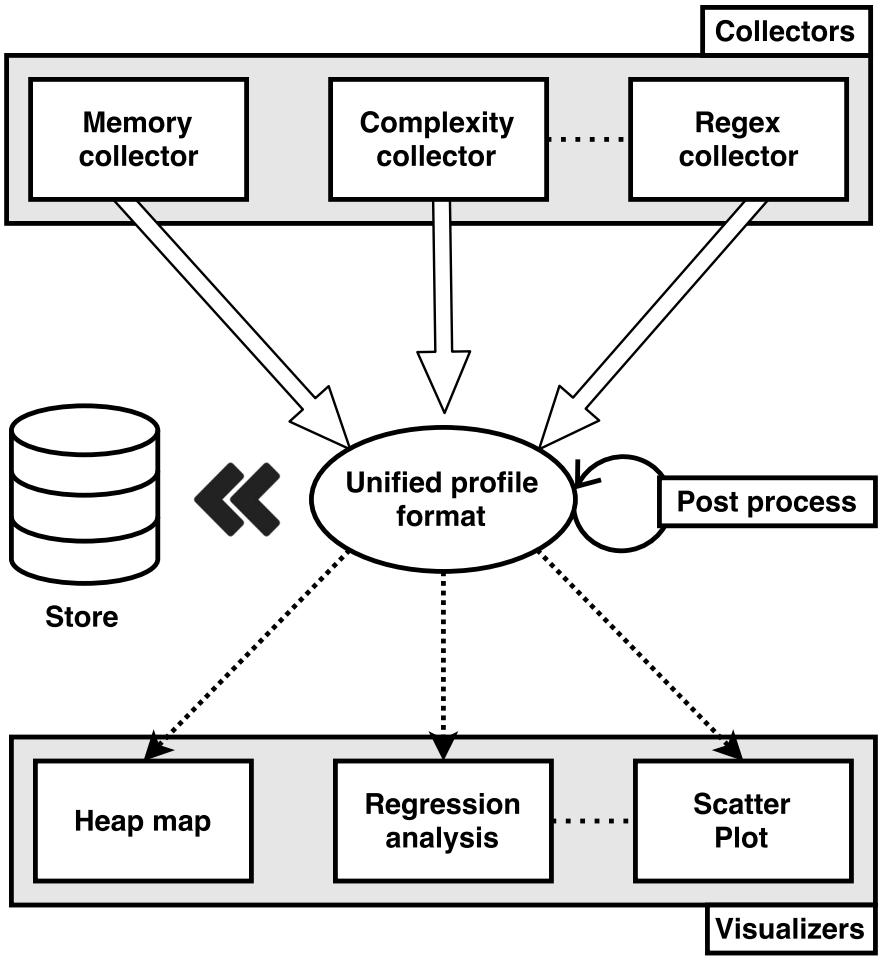
In order to partially verify that Perun runs correctly in your environment, run the automated tests as follows:

```
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*.

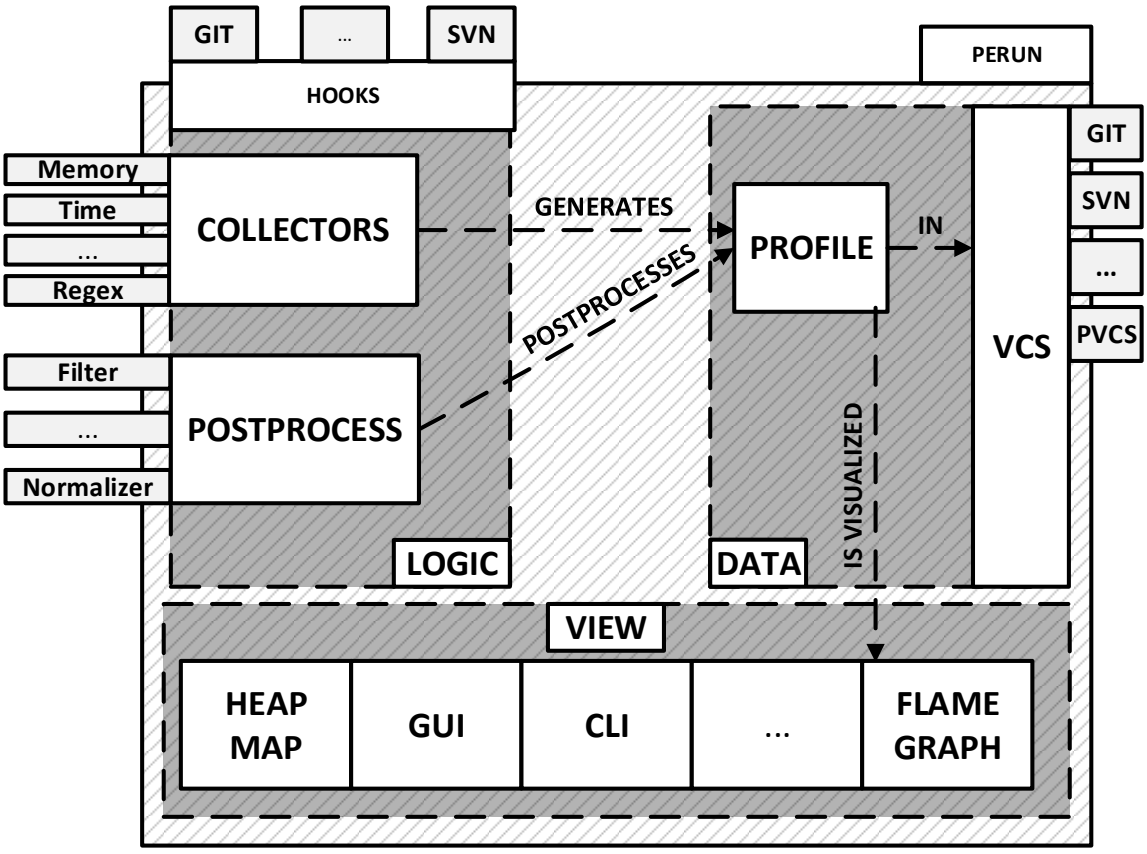


The Figure above shows the lifetime of one profile. Profiles can be generated by set of collectors (such as *Trace Collector* which collects time durations depending on sizes of data structures, or simple *Time Collector* for basic timing) and can be further refined and transformed by sequence of postprocessing 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*, *Creating your own Postprocessor* 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 collector,

profile postprocessor, or new visual interpretation of results refer to *Creating your own Collector*, *Creating your own Postprocessor* 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 *Creating your own Postprocessor*.

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:

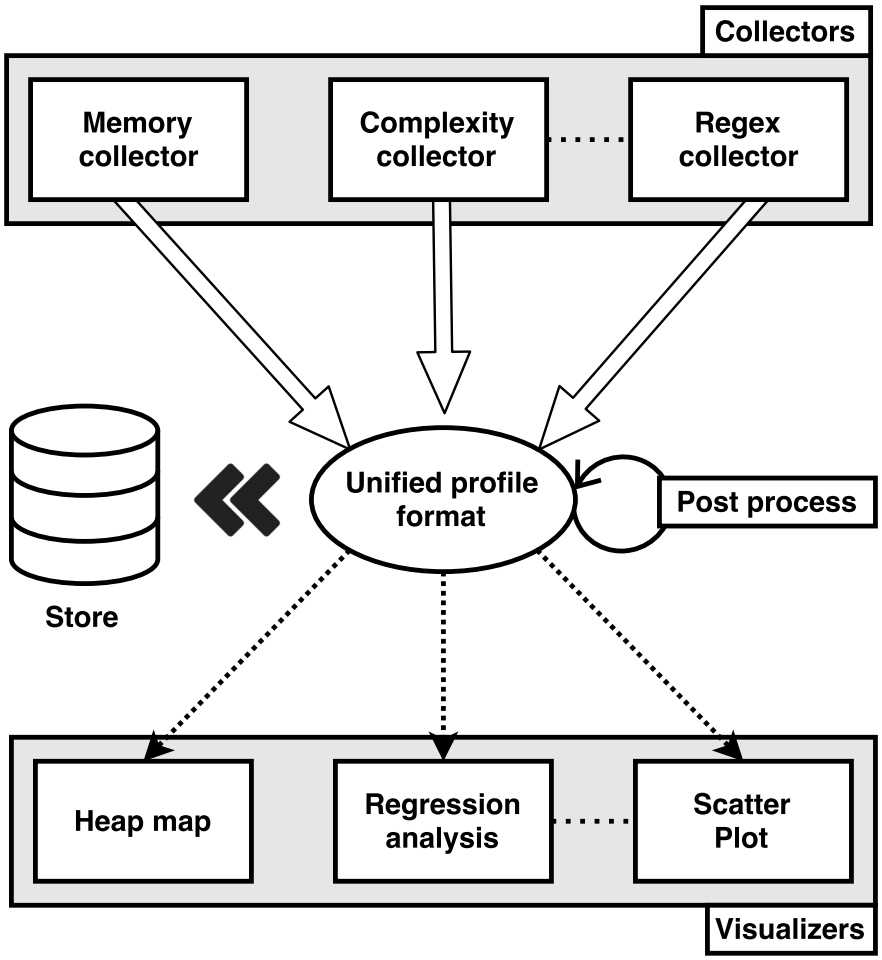
- **Jan Fiedor** (Honeywell)—for feedback, and technical discussions;
- **Martin Hruska** (BUT FIT)—for feedback, and technical discussions;
- **Petr Müller** (SAP)—for nice discussion about the project;
- **Michal Kotoun** (BUT FIT)—for feedback, and having faith in this repo;
- **Hanka Pluhackova** (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",
        "args": "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. *Trace Collector* and will thus omit the actual usage of the command), however, this key can still be used e.g. for tagging the profiles.

args

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 *Trace 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 *Trace 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_start": 0,
    "x_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

Profile factory optimizes the previous profile format

In particular, in the new format we propose to merge some regions into so called resource types, which are dictionaries of persistent less frequently changed aspects of resources. Moreover, we optimize other regions and flatten the format.

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 library 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`** (*`Profile`*) – 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** (*Profile*) – 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** (*Profile*) – 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 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** (*Profile*) – 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_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 occurring in the resources.

E.g. considering the example profiles from `resources`, the function yields the following for *memory*, *time* and *trace* 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** (*Profile*) – 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 *trace* 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** (*Profile*) – 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 *trace* 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** (*Profile*) – 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.**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 trace profile `complexity_prof`, we can obtain unique values of keys from *models* as follows:

```
>>> list(query.unique_model_values_of('model')
['constant', 'exponential', 'linear', 'logarithmic', 'quadratic']
>>> list(query.unique_model_values_of('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** (*Profile*) – 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
Disables the paging of the long standard output (currently affects only `status` and `log` outputs). See `paging` to change the default paging strategy.

-nc, --no-color
Disables the colored output.

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

--version
Prints the current version of Perun.

Commands

add

check

collect

config

fuzz

init

log

postprocessby

rm

run

show

status

utils

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.

`-t, --config-template` `<config_template>`
: States the configuration template that will be used for initialization of local configuration. See *Predefined Configuration Templates* for more details about predefined configurations.

Arguments

`<path>`
: Optional argument

3.2.2 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`, `pending tag range` 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. The `pending tag range` is in form of `i@p-j@p`, where both `i` and `j` stands for indexes in the pending profiles. The `pending tag range` then represents all of the profiles in the interval <i, j>. When `i > j`, then no profiles will be add; when `j`; when `j` is bigger than the number of pending profiles, then all of the non-existing pending profiles will be obviously skipped. Run `perun status` to see the *tag* anotation of pending profiles. Tags consider the sorted order as specified by the following option *format.sort_profiles_by*.

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.

-f, --force
: If set to true, then the profile will be registered in the <hash> minor versionindex, even if its origin <hash> is different. WARNING: This can screw the performance history of your project.

Arguments

<profile>
: Required argument(s)

3.2.3 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`, `pending tag` or a path specifying the profile either in index or in the pending jobs. `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. The `index tag range` is in form of `i@i-j@i`, where both `i` and `j` stands for indexes in the minor version's index. The `index tag range` then represents all of the profiles in the interval <i, j>. registered in index. When `i > j`, then no profiles will be removed; when `j`; when `j` is bigger than the number of pending profiles, then all of the non-existing pending profiles will be obviously skipped. The `pending tags` and `pending tag range` are defined analogously to index tags, except they use the `p` character, i.e. `0@p` and `0@p-2@p` are valid pending tag and pending tag range. Otherwise one can use the path to represent the removed profile. If the path points to existing profile in pending jobs (i.e. `.perun/jobs` directory) the profile is removed from the jobs, otherwise it is looked-up in the index. Tags consider the sorted order as specified by the following option *`format.sort_profiles_by`*.

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).

Arguments

`<profile>`
Required argument(s)

3.2.4 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`).

Profiles (both registered in index and stored in pending directory) are sorted according to the *format.sort_profiles_by*. The option `--sort-by` sets this key in the local configuration for further usage. This means that using the pending or index tags will consider this order.

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 information.

-sb, --sort-by <format__sort_profiles_by>
Sets the <key> in the local configuration for sorting profiles. Note that after setting the <key> it will be used for sorting which is considered in pending and index tags!

3.2.5 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.shortlog* in nearest configuration.

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

Options

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

Arguments

<hash>
Optional argument

3.2.6 perun fuzz

Performs fuzzing for the specified command according to the initial sample of workload.

```
perun fuzz [OPTIONS]
```

Options

-b, --cmd <cmd>
The command which will be fuzzed. [required]

-a, --args <args>
Arguments for the fuzzed command.

-w, --input-sample <input_sample>
Initial sample of workloads, that will be source of the fuzzing. [required]

-c, --collector <collector>
Collector that will be used to collect performance data.

-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

-m, --minor-version <minor_version_list>
Specifies the head minor version, for which the fuzzing will be performed.

-wf, --workloads-filter <workloads_filter>
Regular expression for filtering the workloads.

-s, --source-path <source_path>
The path to the directory of the project source files.

-g, --gcno-path <gcno_path>
The path to the directory where .gcno files are stored.

-o, --output-dir <output_dir>
The path to the directory where generated outputs will be stored. [required]

-t, --timeout <timeout>
Time limit for fuzzing (in seconds). Default value is 1800s.

-h, --hang-timeout <hang_timeout>
The time limit before input is classified as a hang (in seconds). Default value is 30s.

-N, --max <max>
The maximum size limit of the generated input file. Value should be larger than any of the initial workload, otherwise it will be adjusted

-mg, --max-size-gain <max_size_gain>
Max size expressed by gain. Using this option, max size of generated input file will be set to (size of the largest workload + value).Default value is 1 000 000 B = 1MB.

-mp, --max-size-ratio <max_size_ratio>
Max size expressed by percentage. Using this option, max size of generated input file will be set to (size of the largest workload * value). E.g. 1.5, max size=largest workload size * 1.5

-e, --exec-limit <exec_limit>
Defines maximum number executions while gathering interesting inputs.

-l, --interesting-files-limit <interesting_files_limit>
Defines minimum number of gathered interesting inputs before perun testing.

-cr, --coverage-increase-rate <coverage_increase_rate>
Represents threshold of coverage increase against base coverage. E.g 1.5, base coverage = 100 000, so threshold = 150 000.

-mpr, --mutations-per-rule <mutations_per_rule>
Strategy which determines how many mutations will be generated by certain fuzzing rule in one iteration: unitary, proportional, probabilistic, mixed

-r, --regex-rules <regex_rules>
Option for adding custom rules specified by regular expressions, written in YAML format file.

-np, --no-plotting
Avoiding sometimes lengthy plotting of graphs.

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

-m, --minor-version <minor_version_list>
Specifies the head minor version, for which the profiles will be collected.

-cp, --crawl-parents
If set to true, then for each specified minor versions, profiles for parents will be collected as well

-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 trace

Generates *trace* 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
}
```

Trace 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.

The collector interface operates with two seemingly same concepts: (external) command and binary. External command refers to the script, executable, makefile, etc. that will be called / invoked during the profiling, such as 'make test', 'run_script.sh', './my_binary'. Binary, on the other hand, refers to the actual binary or executable file that will be profiled and contains specified functions / static probes etc. It is expected that the binary will be invoked / called as part of the external command script or that external command and binary are the same.

The interface for rules (functions, static probes) specification offers a way to specify profiled locations both with sampling or without it. Note that sampling can reduce the overhead imposed by the profiling. Static rules can be further paired - paired rules act as a start and end point for time measurement. Without a pair, the rule measures time between each two probe hits. The pairing is done automatically for static locations with convention <name> and <name>_end or <name>_END. Otherwise, it is possible to pair rules by the delimiter '#', such as <name1>#<name2>.

Trace 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 trace of individual functions.

Scatter plots are suitable visualization for profiles collected by *trace* 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 *Trace Collector* for more thorough description and examples of *trace* collector.

```
perun collect trace [OPTIONS]
```

Options

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

-f, --func <func>
Set the probe point for the given function.

-s, --static <static>
Set the probe point for the given static location.

-d, --dynamic <dynamic>
Set the probe point for the given dynamic location.

-fs, --func-sampled <func_sampled>
Set the probe point and sampling for the given function.

-ss, --static-sampled <static_sampled>
Set the probe point and sampling for the given static location.

-ds, --dynamic-sampled <dynamic_sampled>
Set the probe point and sampling for the given dynamic location.

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

--with-static, --no-static
The selected method will also extract and profile static probes.

-b, --binary <binary>
The profiled executable. If not set, then the command is considered to be the profiled executable and is used as a binary parameter

-t, --timeout <timeout>
Set time limit (in seconds) for the profiled command, i.e. the command will be terminated after reaching the time limit. Useful for endless commands etc.

-z, --zip-temps
Zip and compress the temporary files (SystemTap log, raw performance data, watchdog log ...) in the perun log directory before deleting them.

-k, --keep-temps
Do not delete the temporary files in the file system.

-vt, --verbose-trace
Set the trace file output to be more verbose, useful for debugging.

-q, --quiet
Reduces the verbosity of the collector info messages.

-w, --watchdog
Enable detailed logging of the whole collection process.

-o, --output-handling `<output_handling>`
Sets the output handling of the profiled command: - default: the output is displayed in the terminal - capture: the output is being captured into a file as well as displayed in the terminal (note that buffering causes a delay in the terminal output) - suppress: redirects the output to the DEVNULL

-i, --diagnostics
Enable detailed surveillance mode of the collector. The collector turns on detailed logging (watchdog), verbose trace, capturing output etc. and stores the logs and files in an archive (zip-temps) in order to provide as much diagnostic data as possible for further inspection.

perun collect memory

Generates *memory* performance profile, 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",
    "subtype": "sys",
    "uid": cmd
    "order": 1
}
```

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

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

Options

-w, --warmup <warmup>
: Before the actual timing, the collector will execute <int> warm-up executions.

-r, --repeat <repeat>
: The timing of the given binaries will be repeated <int> times.

perun collect bounds

Generates *memory* performance profile, 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.

```
{
        "uid": {
            "source": "../test.c",
            "function": "main",
            "line": 22
            "column": 40
        }
        "bound": "1 + max(0, (k + -1))",
        "class": "O(n^1)"
        "type": "bound",
}
```

```
`Memory` profiles can be efficiently interpreted using :ref:`views-heapmap`
technique (together with its `heat` mode), which shows memory allocations
(by functions) in memory address map.

Refer to :ref:`collectors-memory` for more thorough description and
examples of `memory` collector.
```

```
perun collect bounds [OPTIONS]
```

Options

-s, --source, --src <source>
Source C file that will be analyzed.

-d, --source-dir <source_dir>
Directory, where source C files are stored. All of the existing files with valid extensions (.c).

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.

Tags consider the sorted order as specified by the following option *format.sort_profiles_by*.

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**: *Trace 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_start": 0,
    "x_end": 11892,
    "model": "linear",
    "method": "full",
}
```

Note that if your data are not suitable for regression analysis, check out *Clusterizer* to postprocess your profile to be analysable by this analysis.

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

-dp, --depending-on <per_key>
Sets the key that will be used as a source of independent variable.

-o, --of <of_key>
Sets key for which we are finding the model.

perun postprocessby clusterizer

Clusters each resource to an appropriate cluster in order to be postprocessable by regression analysis.

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

Clusterizer tries to find a suitable cluster for each resource in the profile. The clusters are either computed w.r.t the sort order of the resource amounts, or are computed according to the sliding window.

The sliding window can be further adjusted by setting its **width** (i.e. how many near values on the x axis will we fit to a cluster) and its **height** (i.e. how big of an interval of resource amounts will be consider for one cluster). Both **width** and **height** can be further augmented. **Width** can either be *absolute*, where we take in maximum the absolute number of resources, *relative*, where we take in maximum the percentage of number of resources for each cluster, or *weighted*, where we take the number of resource depending on the frequency of their occurrences. Similarly, the **height** can either be *absolute*, where we set the interval of amounts to an absolute size, or *relative*, where we set the interval of amounts relative to the to the first resource amount in the cluster (so e.g. if we have window of height 0.1 and the first resource in the cluster has amount of 100, we will cluster every resources in interval 100 to 110 to this cluster).

For more details about regression analysis refer to *Clusterizer*.

```
perun postprocessby clusterizer [OPTIONS]
```

Options

-s, --strategy <strategy>
Specifies the clustering strategy, that will be applied for the profile

-wh, --window-height <window_height>
Specifies the height of the window (either fixed or proportional)

-rwh, --relative-window-height
Specifies that the height of the window is relative to the point

-fwh, --fixed-window-height
Specifies that the height of the window is absolute to the point

-ww, --window-width <window_width>
Specifies the width of the window, i.e. how many values will be taken by window.

-rww, --relative-window-width
Specifies whether the width of the window is weighted or fixed

-fww, --fixed-window-width
Specifies whether the width of the window is weighted or fixed

-www, --weighted-window-width
Specifies whether the width of the window is weighted or fixed

perun postprocessby regressogram

Execution of the interleaving of profiled resources by **regressogram** models.

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

Regressogram belongs to the simplest non-parametric methods and its properties are the following:

> **Regressogram**: can be described such as step function (i.e. constant function by parts). Regressogram uses the same basic idea as a histogram for density estimate. This idea is in dividing the set of values of the x-coordinates (<*per_key*>) into intervals and the estimate of the point in concrete interval takes the mean/median of the y-coordinates (<*of_resource_key*>), respectively of its value on this sub-interval. We currently use the *coefficient of determination* (R^2) to measure the fitness of regressogram. The fitness of estimation of regressogram model depends primarily on the number of buckets into which the interval will be divided. The user can choose number of buckets manually (<*bucket_window*>) or use one of the following methods to estimate the optimal number of buckets (<*bucket_method*>):
>
> - **sqrt**: square root (of data size) estimator, used for its speed and simplicity
> - **rice**: does not take variability into account, only data size and commonly overestimates
> - **scott**: takes into account data variability and data size, less robust estimator
> - **stone**: based on leave-one-out cross validation estimate of the integrated squared error
> - **fd**: robust, takes into account data variability and data size, resilient to outliers
> - **sturges**: only accounts for data size, underestimates for large non-gaussian data
> - **doane**: generalization of Sturges' formula, works better with non-gaussian data
> - **auto**: max of the Sturges' and 'fd' estimators, provides good all around performance
>
> For more details about these methods to estimate the optimal number of buckets or to view the code of these methods, you can visit SciPy.

For more details about this approach of non-parametric analysis refer to *Regressogram method*.

```
perun postprocessby regressogram [OPTIONS]
```

Options

-bn, --bucket_number <bucket_number>
: Restricts the number of buckets to which will be placed the values of the selected statistics.

-bm, --bucket_method <bucket_method>
: Specifies the method to estimate the optimal number of buckets.

-sf, --statistic_function <statistic_function>
: Will use the <statistic_function> to compute the values for points within each bucket of regressogram.

-of, --of-key <of_key>
: Sets key for which we are finding the model (y-coordinates).

-per, --per-key <per_key>
: Sets the key that will be used as a source variable (x-coordinates).

perun postprocessby moving_average

Execution of the interleaving of profiled resources by *moving average* models.

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

Moving average methods are the natural generalizations of regressogram method. This method uses the local averages/medians of y-coordinates (<*of_resource_key*>), but the estimate in the x-point (<*per_key*>) is based on the centered surroundings of this points, more precisely:

> **Moving Average**: is a widely used estimator in the technical analysis, that helps smooth the dataset by filtering out the 'noise'. Among the basic properties of this methods belongs the ability to reduce the effect of temporary variations in data, better improvement of the fitness of data to a line, so called smoothing, to show the data's trend more clearly and highlight any value below or above the trend. The most important task with this type of non-parametric approach is the choice of the <*window-width*>. If the user does not choose it, we try approximate this value by using the value of *coefficient of determination* (R^2). At the begin of the analysis is set the initial value of window width and then follows the interleaving of the current dataset, which runs until the value of *coefficient of determination* will not reach the required level. By this way is guaranteed the desired smoothness of the resulting models. The two basic and commonly used <*moving-methods*> are the **simple** moving average (**sma**) and the *exponential* moving average (**ema**).

For more details about this approach of non-parametric analysis refer to *Moving Average Methods*.

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

Options

-mp, --min_periods <min_periods>
Provides the minimum number of observations in window required to have a value. If the number of possible observations smaller then result is NaN.

-of, --of-key <of_key>
Sets key for which we are finding the model (y-coordinates).

-per, --per-key <per_key>
Sets the key that will be used as a source variable (x-coordinates).

Commands

ema

sma

smm

perun postprocessby moving_average sma

Simple Moving Average

In the most of cases, it is an unweighted Moving Average, this means that the each x-coordinate in the data set (profiled resources) has equal importance and is weighted equally. Then the *mean* is computed from the previous *n data* (<*no-center*>), where the *n* marks <*window-width*>. However, in science and engineering the mean is normally taken from an equal number of data on either side of a central value (<*center*>). This ensures that variations in the mean are aligned with the variations in the mean are aligned with variations in the data rather than being shifted in the x-axis direction. Since the window at the boundaries of the interval does not contain enough count of points usually, it is

necessary to specify the value of <*min-periods*> to avoid the NaN result. The role of the weighted function in this approach belongs to <*window-type*>, which represents the suite of the following window functions for filtering:

- **boxcar**: known as rectangular or Dirichlet window, is equivalent to no window at all: –
- **triang**: standard triangular window
- **blackman**: formed by using three terms of a summation of cosines, minimal leakage, close to optimal
- **hamming**: formed by using a raised cosine with non-zero endpoints, minimize the nearest side lobe
- **bartlett**: similar to triangular, endpoints are at zero, processing of tapering data sets
- **parzen**: can be regarded as a generalization of k-nearest neighbor techniques
- **bohman**: convolution of two half-duration cosine lobes
- **blackmanharris**: minimum in the sense that its maximum side lobes are minimized (symmetric 4-term)
- **nuttall**: minimum 4-term Blackman-Harris window according to Nuttall (so called 'Nuttall4c')
- **barthann**: has a main lobe at the origin and asymptotically decaying side lobes on both sides
- **kaiser**: formed by using a Bessel function, needs beta value (set to 14 - good starting point)

For more details about this window functions or for their visual view you can see SciPyWindow.

```
perun postprocessby moving_average sma [OPTIONS]
```

Options

-wt, --window_type <window_type>
: Provides the window type, if not set then all points are evenly weighted. For further information about window types see the notes in the documentation.

--center, --no-center
: If set to False, the result is set to the right edge of the window, else is result set to the center of the window

-ww, --window_width <window_width>
: Size of the moving window. This is a number of observations used for calculating the statistic. Each window will be a fixed size.

perun postprocessby moving_average smm

Simple Moving Median

The second representative of Simple Moving Average methods is the Simple Moving **Median**. For this method are applicable to the same rules like in the first described method, except for the option for choosing the window type, which do not make sense in this approach. The only difference between these two methods are the way of computation the values in the individual sub-intervals. Simple Moving **Median** is not based on the computation of average, but as the name suggests, it based on the **median**.

```
perun postprocessby moving_average smm [OPTIONS]
```

Options

--center, --no-center
: If set to False, the result is set to the right edge of the window, else is result set to the center of the window

-ww, --window_width <window_width>
: Size of the moving window. This is a number of observations used for calculating the statistic. Each window will be a fixed size.

perun postprocessby moving_average ema

Exponential Moving Average

This method is a type of moving average methods, also know as **Exponential** Weighted Moving Average, that places a greater weight and significance on the most recent data points. The weighting for each far x-coordinate decreases exponentially and never reaching zero. This approach of moving average reacts more significantly to recent changes than a *Simple* Moving Average, which applies an equal weight to all observations in the period. To calculate an EMA must be first computing the **Simple** Moving Average (SMA) over a particular sub-interval. In the next step must be calculated the multiplier for smoothing (weighting) the EMA, which depends on the selected formula, the following options are supported (<*decay*>):

- **com**: specify decay in terms of center of mass: α = 1 / (1 + com), for com >= 0
- **span**: specify decay in terms of span: α = 2 / (span + 1), for span >= 1
- **halflife**: specify decay in terms of half-life, α = 1 - exp(log(0.5) / halflife), for halflife > 0
- **alpha**: specify smoothing factor α directly: 0 < α <= 1

The computed coefficient α represents the degree of weighting decrease, a constant smoothing factor, The higher value of α discounts older observations faster, the small value to the contrary. Finally, to calculate the current value of EMA is used the relevant formula. It is important do not confuse **Exponential** Moving Average with **Simple** Moving Average. An **Exponential** Moving Average behaves quite differently from the second mentioned method, because it is the function of weighting factor or length of the average.

```
perun postprocessby moving_average ema [OPTIONS]
```

Options

-d, --decay <decay>
Exactly one of “com”, “span”, “halflife”, “alpha” can be provided. Allowed values and relationship between the parameters are specified in the documentation (e.g. –decay=com 3).

perun postprocessby kernel-regression

Execution of the interleaving of profiles resources by *kernel* models.

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

In statistics, the kernel regression is a non-parametric approach to estimate the conditional expectation of a random variable. Generally, the main goal of this approach is to find non-parametric relation between a pair of random variables X <per-key> and Y <of-key>. Different from parametric techniques (e.g. linear regression), kernel regression does not assume any underlying distribution (e.g. linear, exponential, etc.) to estimate the regression function. The main idea of kernel regression is putting the **kernel**, that have the role of weighted function, to each observation point in the dataset. Subsequently, the kernel will assign weight to each point in depends on the distance from the current data point. The kernel basis formula depends only to the *bandwidth* from the current (‘local’) data point X to a set of neighboring data points X.

> **Kernel Selection** does not important from an asymptotic point of view. It is appropriate to choose the **optimal** kernel since this group of the kernels are continuously on the whole definition field and then the estimated regression function inherit smoothness of the kernel. For example, a suitable kernels can be the **epanechnikov** or **normal** kernel. This postprocessor offers the **kernel selection** in the **kernel-smoothing** mode, where are available five different types of kernels. For more information about these kernels or this kernel regression mode you can see *perun postprocessby kernel-regression kernel-smoothing*.

Bandwidth Selection is the most important factor at each approach of kernel regression, since this value significantly affects the smoothness of the resulting estimate. In case, when is choose the inappropriate value, in the most cases can be expected the following two situations. The **small** bandwidth value reproduce estimated data and vice versa, the **large** value leads to over-leaving, so to average of the estimated data. Therefore are used the methods to determine the bandwidth value. One of the most widespread and most commonly used methods is the **cross-validation** method. This method is based on the estimate of the regression function in which will be omitted *i-th* observation. In this postprocessor is this method available in the **estimator-setting** mode. Another methods to determine the bandwidth, which are available in the remaining modes of this postprocessor are **scott** and **silverman** method. More information about these methods and its definition you cas see in the part *perun postprocessby kernel-regression method-selection*.

This postprocessor in summary offers five different modes, which does not differ in the resulting estimate, but in the way of computation the resulting estimate. Better said, it means, that the result of each mode is the **kernel estimate** with relevant parameters, selected according to the concrete mode. In short we will describe the individual methods, for more information about it, you can visit the relevant parts of documentation:

* **Estimator-Settings**: Nadaraya-Watson kernel regression with specific settings for estimate
* **User-Selection**: Nadaraya-Watson kernel regression with user bandwidth
* **Method-Selection**: Nadaraya-Watson kernel regression with supporting bandwidth selection method
* **Kernel-Smoothing**: Kernel regression with different types of kernel and regression methods
* **Kernel-Ridge**: Nadaraya-Watson kernel regression with automatic bandwidth selection

For more details about this approach of non-parametric analysis refer to *Kernel Regression Methods*.

```
perun postprocessby kernel-regression [OPTIONS] COMMAND [ARGS]...
```

Options

-of, --of-key <of_key>
Sets key for which we are finding the model (y-coordinates).

-per, --per-key <per_key>
Sets the key that will be used as a source variable (x-coordinates).

Commands

estimator-settings

kernel-ridge

kernel-smoothing

method-selection

user-selection

perun postprocessby kernel-regression estimator-settings

Nadaraya-Watson kernel regression with specific settings for estimate.

As has been mentioned above, the kernel regression aims to estimate the functional relation between explanatory variable **y** and the response variable **X**. This mode of kernel regression postprocessor calculates the conditional mean **E[y|X] = m(X)**, where **y = m(X) +** ϵ. Variable **X** is represented in the postprocessor by <per-key> option and the variable **y** is represented by <of-key> option.

Regression Estimator <reg-type>:

This mode offer two types of *regression estimator <reg-type>*. *Local Constant ('ll')* type of regression provided by this mode is also known as *Nadaraya-Watson* kernel regression:

Nadaraya-Watson: expects the following conditional expectation: **E[y|X] = m(X)**, where function **m(*)** represents the regression function to estimate. Then we can alternatively write the following formula: **y = m(X) +** ϵ, **E** (ϵ) **= 0**. Then we can suppose, that we have the set of independent observations $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ and the **Nadaraya-Watson** estimator is defined as:

$$m_h(x) = \sum_{i=1}^{n} K_h(x - x_i) y_i / \sum_{j=1}^{n} K_h(x - x_j)$$

where K_h is a kernel with bandwidth *h*. The denominator is a weighting term with sum 1. It easy to see that this kernel regression estimator is just a weighted sum of the observed responses y_i. There are many other kernel estimators that are various in compare to this presented estimator. However, since all are asymptotic equivalently, we will not deal with them closer. **Kernel Regression** postprocessor works in all modes only with **Nadaraya-Watson** estimator.

The second supported *regression estimator* in this mode of postprocessor is *Local Linear ('lc')*. This type is an extension of that which suffers less from bias issues at the edge of the support.

Local Linear: estimator, that offers various advantages compared with other kernel-type estimators, such as the *Nadaraya-Watson* estimator. More precisely, it adapts to both random and fixed designs, and to various design densities such as highly clustered designs and nearly uniform designs. It turns out that the *local linear* smoother repairs the drawbacks of other kernel regression estimators. An regression estimator *m* of *m* is a linear smoother if, for each *x*, there is a vector $l(x) = (l_1(x), ..., l_n(x))^T$ such that:

$$m(x) = \sum_{i=1}^{n} l_i(x) Y_i = l(x)^T Y$$

where $Y = (Y_1, ..., Y_n)^T$. For kernel estimators:

$$l_i(x) = K(||x - X_i||/h) / \sum_{j=1}^{n} K(||x - X_j||/h)$$

where *K* represents kernel and *h* its bandwidth.

For a better imagination, there is an interesting fact, that the following estimators are linear smoothers too: *Gaussian process regression*, *splines*.

Bandwidth Method <bandwidth-method>:

As has been said in the general description of the *kernel regression*, once of the most important factors of the resulting estimate is the kernel **bandwidth**. When the inappropriate value is selected may occur to *under-laying* or *over-laying* fo the resulting kernel estimate. Since the bandwidth of the kernel is a free parameter which exhibits a strong influence on the resulting estimate postprocessor offers the method for its selection. Two most popular data-driven methods of bandwidth selection that have desirable properties are *least-squares cross-validation* (*cv_ls*) and the *AIC-based* method of *Hurvich et al. (1998)*, which is based on minimizing a modified *Akaike Information Criterion* (*aic*):

Cross-Validation Least-Squares: determination of the optimal kernel bandwidth for kernel regression is based on minimizing

$$CV(h) = n^{-1} \sum_{i=1}^{n} (Y_i - g_{-i}(X_i))^2,$$

where $g_{-i}(X_i)$ is the estimator of $g(X_i)$ formed by leaving out the *i-th* observation when generating the prediction for observation *i*.

Hurvich et al.'s (1998) approach is based on the minimization of

$$AIC_c = ln(\sigma^2) + ((1 + tr(H)/n)/(1 - (tr(H) + 2)/n),$$

where

$$\sigma^2 = 1/n \sum_{i=1}^{n} (Y_i - g(X_i))^2 = Y'(I - H)'(I - H)Y/n$$

with $g(X_i)$ being a non-parametric regression estimator and *H* being an *n x n* matrix of kernel weights with its *(i, j)-th* element given by $H_{ij} = K_h(X_i, X_j)/\sum_{l=1}^{n} K_h(X_i, X_l)$, where $K_h(*)$ is a generalized product kernel.

Both methods for kernel bandwidth selection the *least-squared cross-validation* and the *AIC* have been shown to be asymptotically equivalent.

The remaining options at this mode of kernel regression postprocessor are described within usage to it and you can see this in the list below. All these options are parameters to *EstimatorSettings* (see EstimatorSettings), that optimizing the kernel bandwidth based on the these specified settings.

In the case of confusion about this approach of kernel regression, you can visit StatsModels.

```
perun postprocessby kernel-regression estimator-settings [OPTIONS]
```

Options

-rt, --reg-type <reg_type>
: Provides the type for regression estimator. Supported types are: "lc": local-constant (Nadaraya-Watson) and "ll": local-linear estimator. Default is "ll". For more information about these types you can visit Perun Documentation.

-bw, --bandwidth-method <bandwidth_method>
: Provides the method for bandwidth selection. Supported values are: "cv-ls": least-squares cross validation and "aic": AIC Hurvich bandwidth estimation. Default is "cv-ls". For more information about these methods you can visit Perun Documentation.

--efficient, --uniformly
: If True, is executing the efficient bandwidth estimation - by taking smaller sub-samples and estimating the scaling factor of each sub-sample. It is useful for large samples and/or multiple variables. If False (default), all data is used at the same time.

--randomize, --no-randomize
: If True, the bandwidth estimation is performed by taking <n_res> random re-samples of size <n-sub-samples> from the full sample. If set to False (default), is performed by slicing the full sample in sub-samples of <n-sub-samples> size, so that all samples are used once.

-nsub, --n-sub-samples <n_sub_samples>
: Size of the sub-samples (default is 50).

-nres, --n-re-samples <n_re_samples>
The number of random re-samples used to bandwidth estimation. It has effect only if <randomize> is set to True. Default values is 25.

--return-median, --return-mean
If True, the estimator uses the median of all scaling factors for each sub-sample to estimate bandwidth of the full sample. If False (default), the estimator used the mean.

perun postprocessby kernel-regression user-selection

Nadaraya-Watson kernel regression with user bandwidth.

This mode of kernel regression postprocessor is very similar to *estimator-settings* mode. Also offers two types of *regression estimator* <reg-type> and that the *Nadaraya-Watson* estimator, so known as *local-constant* (*lc*) and the *local-linear* estimator (*ll*). Details about these estimators are available in *perun postprocessby kernel-regression estimator-settings*. In contrary to this mode, which selected a kernel bandwidth using the *EstimatorSettings* and chosen parameters, in this mode the user itself selects a kernel bandwidth <bandwidth-value>. This value will be used to execute the kernel regression. The value of kernel bandwidth in the resulting estimate may change occasionally, specifically in the case, when the bandwidth value is too low to execute the kernel regression. Then will be a bandwidth value approximated to the closest appropriate value, so that is not decreased the accuracy of the resulting estimate.

```
perun postprocessby kernel-regression user-selection [OPTIONS]
```

Options

-rt, --reg-type <reg_type>
Provides the type for regression estimator. Supported types are: “lc”: local-constant (Nadaraya-Watson) and “ll”: local-linear estimator. Default is “ll”. For more information about these types you can visit Perun Documentation.

-bv, --bandwidth-value <bandwidth_value>
The float value of <bandwidth> defined by user, which will be used at kernel regression. [required]

perun postprocessby kernel-regression method-selection

Nadaraya-Watson kernel regression with supporting bandwidth selection method.

The last method from a group of three methods based on a similar principle. *Method-selection* mode offers the same type of *regression estimators* <reg-type> as the first two described methods. The first supported option is *ll*, which represents the *local-linear* estimator. *Nadaraya-Watson* or *local constant* estimator represents the second option for <reg-type> parameter. The more detailed description of these estimators is located in *perun postprocessby kernel-regression estimator-settings*. The difference between this mode and the two first modes is in the way of determination of a kernel bandwidth. In this mode are offered two methods to determine bandwidth. These methods try calculated an optimal bandwidth from predefined formulas:

Scotts's Rule of thumb to determine the smoothing bandwidth for a kernel estimation. It is very fast compute. This rule was designed for density estimation but is usable for kernel regression too. Typically produces a larger bandwidth and therefore it is useful for estimating a gradual trend:

$$bw = 1.059 * A * n^{-1/5},$$

where *n* marks the length of X variable <per-key> and

$$A = min(\sigma(x), IQR(x)/1.349),$$

where σ marks the StandardDeviation and IQR marks the InterquartileRange.

Silverman's Rule of thumb to determine the smoothing bandwidth for a kernel estimation. Belongs to most popular method which uses the *rule-of-thumb*. Rule is originally designs for *density estimation* and therefore uses the normal density as a prior for approximating. For the necessary estimation of the σ of X <per-key> he proposes a robust version making use of the InterquartileRange. If the true density is uni-modal, fairly symmetric and does not have fat tails, it works fine:

$$bw = 0.9 * A * n^{-1/5},$$

where *n* marks the length of X variable <per-key> and

$$A = min(\sigma(x), IQR(x)/1.349),$$

where σ marks the StandardDeviation and IQR marks the InterquartileRange.

```
perun postprocessby kernel-regression method-selection [OPTIONS]
```

Options

-rt, --reg-type <reg_type>
Provides the type for regression estimator. Supported types are: "lc": local-constant (Nadaraya-Watson) and "ll": local-linear estimator. Default is "ll". For more information about these types you can visit Perun Documentation.

-bm, --bandwidth-method <bandwidth_method>
Provides the helper method to determine the kernel bandwidth. The <method_name> will be used to compute the bandwidth, which will be used at kernel regression.

perun postprocessby kernel-regression kernel-smoothing

Kernel regression with different types of kernel and regression methods.

This mode of kernel regression postprocessor implements non-parametric regression using different kernel methods and different kernel types. The calculation in this mode can be split into three parts. The first part is represented by the *kernel type*, the second part by *bandwidth computation* and the last part is represented by *regression method*, which will be used to interleave the given resources. We will look gradually at individual supported options in the each part of computation.

Kernel Type <kernel-type>:

In non-parametric statistics a *kernel* is a weighting function used in estimation techniques. In *kernel regression* is used to estimate the conditional expectation of a random variable. As has been said, *kernel width* must be specified when running a non-parametric estimation. The *kernel* in view of mathematical definition is a non-negative real-valued integrable function *K*. For most applications, it is desirable to define the function to satisfy two additional requirements:

Normalization:

$$\int_{-\infty}^{+\infty} K(u)du = 1,$$

Symmetry

$$K(-u) = K(u),$$

for all values of u. The second requirement ensures that the average of the corresponding distribution is equal to that of the sample used. If *K* is a kernel, then so is the function K^* defined by $K^*(u) = \lambda K(\lambda u)$, where $\lambda > 0$. This can be used to select a scale that is appropriate for the data. This mode offers several types of kernel functions:

Kernel Name	Kernel Function, K(u)	Efficiency
Gaussian (normal)	$K(u) = (1/\sqrt{2\pi})e^{-(1/2)u^2}$	95.1%
Epanechnikov	$K(u) = 3/4(1-u^2)$	100%
Tricube	$K(u) = 70/81(1-\lvert u^3\rvert)^3$	99.8%
Gaussian order4	$\phi_4(u) = 1/2(3-u^2)\phi(u)$, where ϕ is the normal kernel	not applicable
Epanechnikov order4	$K_4(u) = -(15/8)u^2 + (9/8)$, where *K* is the non-normalized Epanechnikov kernel	not applicable

Efficiency is defined as $\sqrt{\int u^2 K(u)du} \int K(u)^2 du$, and its measured to the *Epanechnikov* kernel.

Smoothing Method <smoothing-method>:

Kernel-Smoothing mode of this postprocessor offers three different non-parametric regression methods to execute *kernel regression*. The first of them is called *spatial-average* and perform a *Nadaraya-Watson* regression (i.e. also called local-constant regression) on the data using a given kernel:

$$m_h(x) = \sum_{i=1}^{n} K_h((x-x_i)/h)y_i / \sum_{j=1}^{n} K_h((x-x_j)/h),$$

where *K(x)* is the kernel and must be such that *E(K(x)) = 0* and *h* is the bandwidth of the method. *Local-Constant* regression was also described in *perun postprocessby kernel-regression estimator-settings*. The second supported regression method by this mode is called *local-linear*. Compared with previous method, which offers computational with different types of kernel, this method has restrictions and perform *local-linear* regression using only *Gaussian (Normal)* kernel. The *local-constant* regression was described in *perun postprocessby kernel-regression estimator-settings* and therefore will not be given no further attention to it. *Local Polynomial* regression is the last method in this mode and perform regression in *N-D* using a user-provided kernel. The *local-polynomial* regression is the function that minimizes, for each position:

$$m_h(x) = \sum_{i=0}^{n} K((x-x_i)/h)(y_i - a_0 - P_q(x_i - x))^2,$$

where *K(x)* is the kernel such that *E(K(x)) = 0*, *q* is the order of the fitted polynomial <polynomial-order>, $P_q(x)$ is a polynomial or order *q* in *x*, and *h* is the bandwidth of the method. The polynomial $P_q(x)$ is of the form:

$$F_d(k) = n \in N^d | \sum_{i=1}^{d} n_i = k$$

$$P_q(x_1, ..., x_d) = \sum_{k=1}^{q} \sum_{n \in F_d(k)} a_{k,n} \prod_{i=1}^{d} x_i^{n_i}$$

For example we can have:

$$P_2(x, y) = a_{110}x + a_{101}y + a_{220}x^2 + a_{221}xy + a_{202}y^2$$

The last part of the calculation is the *bandwidth* computation. This mode offers to user enter the value directly with use of parameter <bandwidth-value>. The parameter <bandwidth-method> offers to user the selection from the two methods to determine the optimal bandwidth value. The supported methods are *Scotts's Rule* and *Silverman's Rule*, which are described in *perun postprocessby kernel-regression method-selection*. This parameter cannot be entered in combination with <bandwidth-value>, then will be ignored and will be accepted value from <bandwidth-value>.

```
perun postprocessby kernel-regression kernel-smoothing [OPTIONS]
```

Options

-kt, --kernel-type <kernel_type>
Provides the set of kernels to execute the *kernel-smoothing* with kernel selected by the user. For exact definitions of these kernels and more information about it, you can visit the Perun Documentation.

-sm, --smoothing-method <smoothing_method>
Provides kernel smoothing methods to executing non-parametric regressions: *local-polynomial* perform a local-polynomial regression in N-D using a user-provided kernel; *local-linear* perform a local-linear regression using a gaussian (normal) kernel; and *spatial-average* perform a Nadaraya-Watson regression on the data (so called local-constant regression) using a user-provided kernel.

-bm, --bandwidth-method <bandwidth_method>
Provides the helper method to determine the kernel bandwidth. The <bandwidth_method> will be used to compute the bandwidth, which will be used at kernel-smoothing regression. Cannot be entered in combination with <bandwidth-value>, then will be ignored and will be accepted value from <bandwidth-value>.

-bv, --bandwidth-value <bandwidth_value>
The float value of <bandwidth> defined by user, which will be used at kernel regression. If is entered in the combination with <bandwidth-method>, then method will be ignored.

-q, --polynomial-order <polynomial_order>
Provides order of the polynomial to fit. Default value of the order is equal to 3. Is accepted only by *local-polynomial* <smoothing-method>, another methods ignoring it.

perun postprocessby kernel-regression kernel-ridge

Nadaraya-Watson kernel regression with automatic bandwidth selection.

This mode implements *Nadaraya-Watson* kernel regression, which was described above in *perun postprocessby kernel-regression estimator-settings*. While the previous modes provided the methods to determine the optimal bandwidth with different ways, this method provides a little bit different way. From a given range of potential bandwidths <gamma-range> try to select the optimal kernel bandwidth with use of *leave-one-out cross-validation*. This approach was described in *perun postprocessby kernel-regression estimator-settings*, where was introduced the *least-squares cross- validation* and it is a modification of this approach. *Leave-one-out cross validation* is K-fold cross validation taken to its logical extreme, with *K* equal to *N*, the number of data points in the set. The original *gamma-range* will be divided on the base of size the given step <gamma-step>. The selection of specific value from this range will be executing by minimizing mean-squared-error in *leave-one-out cross-validation*. The selected *bandwidth-value* will serves for *gaussian* kernel in resulting estimate: $K(x, y) = exp(-gamma * ||x - y||^2)$.

```
perun postprocessby kernel-regression kernel-ridge [OPTIONS]
```

Options

-gr, --gamma-range <gamma_range>
: Provides the range for automatic bandwidth selection of the kernel via leave-one-out cross-validation. One value from these range will be selected with minimizing the mean-squared error of leave-one-out cross-validation. The first value will be taken as the lower bound of the range and cannot be greater than the second value.

-gs, --gamma-step <gamma_step>
: Provides the size of the step, with which will be executed the iteration over the given <gamma-range>. Cannot be greater than length of <gamma-range>, else will be set to value of the lower bound of the <gamma_range>.

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.

Tags consider the sorted order as specified by the following option *format.sort_profiles_by*.

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 bars

Customizable interpretation of resources using the bar format.

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

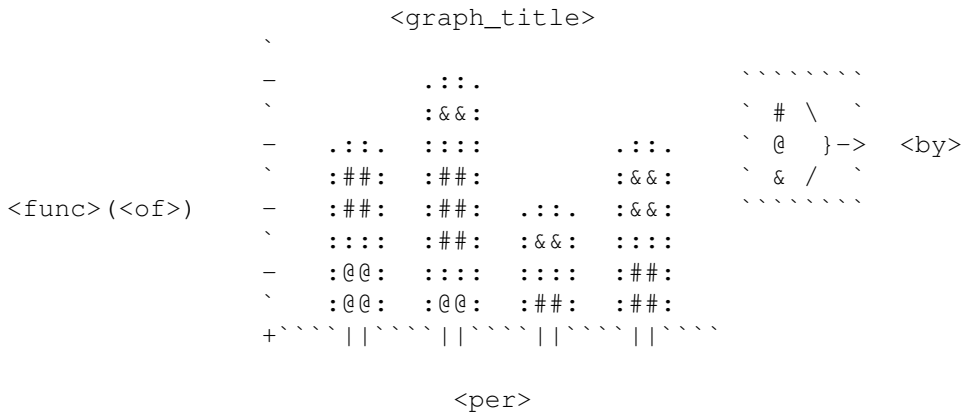
Bars graph shows the aggregation (e.g. sum, count, etc.) of resources of given types (or keys). Each bar shows <func> of resources from <of> key (e.g. sum of amounts, average of amounts, count of types, etc.) per each <per> key (e.g. per each snapshot, or per each type). Moreover, the graphs can either 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

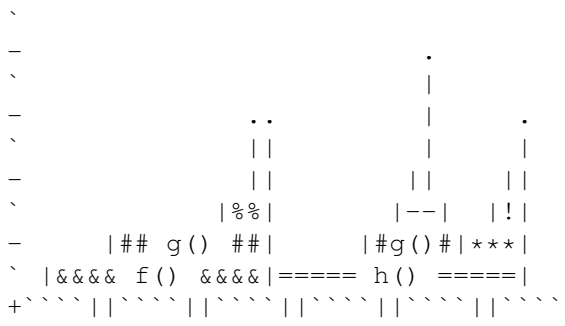
Flame graph interprets the relative and inclusive presence of the resources according to the stack depth of the origin of resources.

- **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

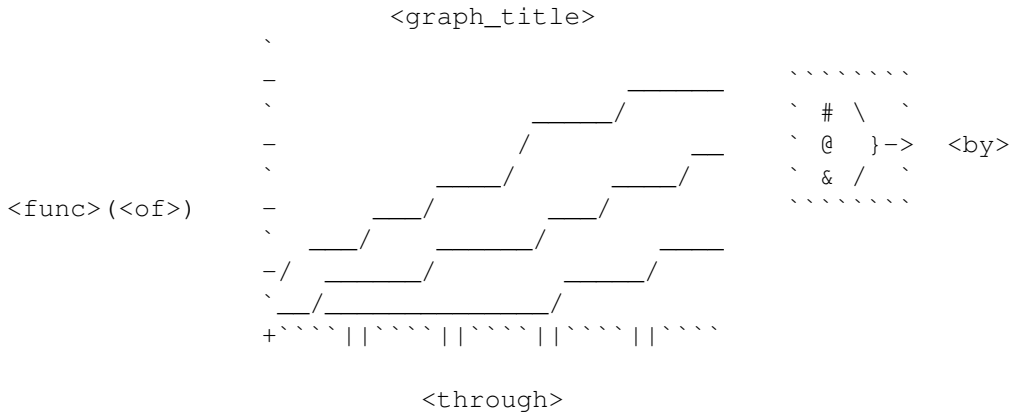
Flow graph shows the values resources depending on the independent variable as basic graph. For each group of resources identified by unique value of <by> key, one graph shows the dependency of <of> values aggregated by <func> depending on the <through> key. Moreover, the values can either be accumulated (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*.

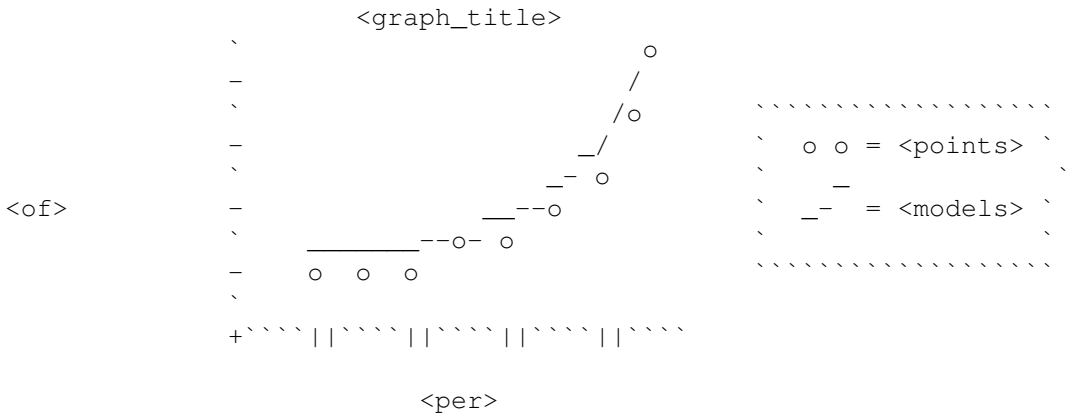
- **Interpretation style**: graphical
- **Visualization backend**: Bokeh

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

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:

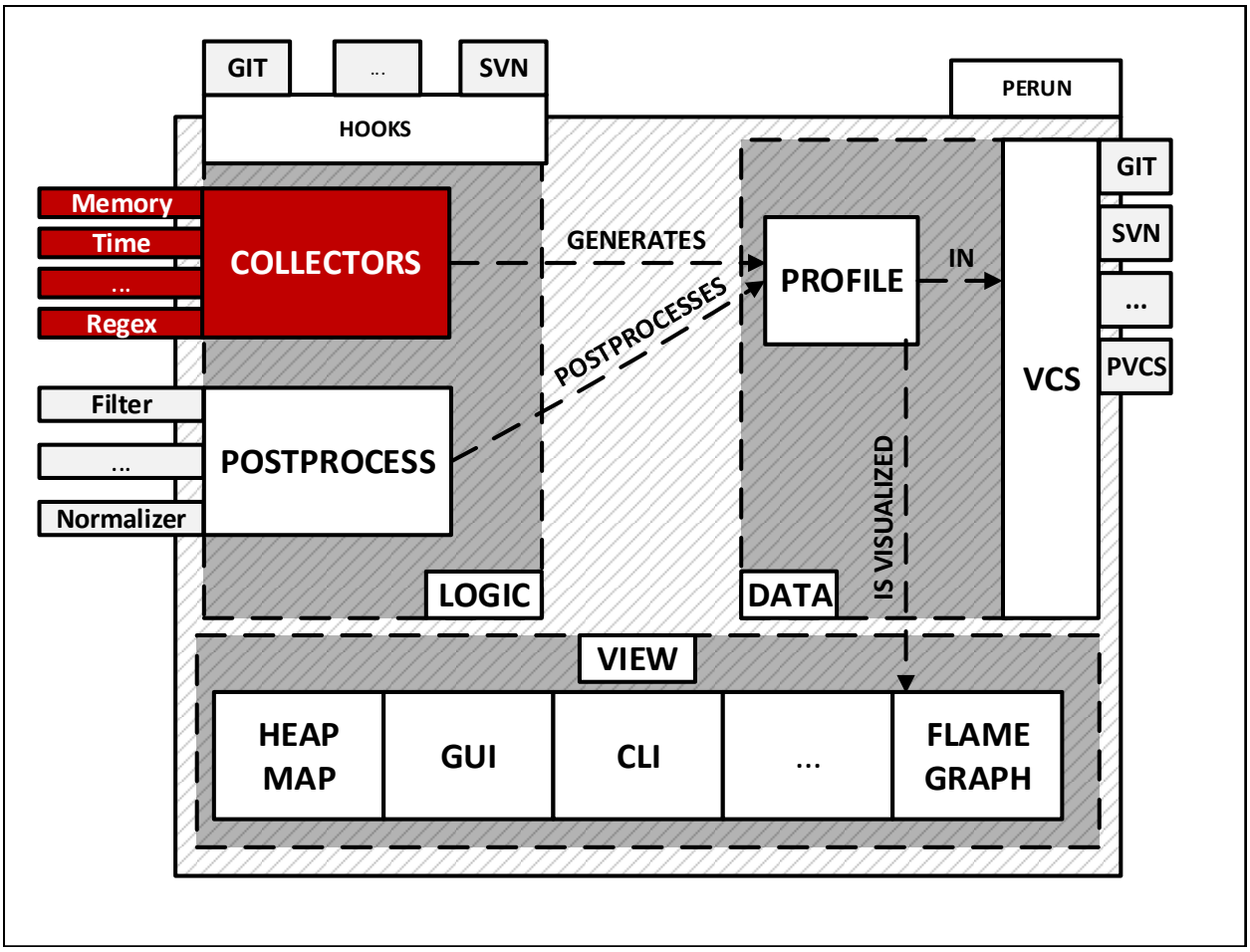
1. By **Directly running collectors** through `perun collect` command, that generates profile using a single collector with given collector configuration. The resulting profiles are not postprocessed in any way.
2. By **Using job specification** either as a single run of batch of profiling jobs using `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. *Trace 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 *Trace 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
4. *Bounds Collector*, collects bounds of integer and, to some extent, heap-manipulating loops represented as so called ranking function. The collectors works as a wrapper over the **Loopus_** tool. The collection is limited to source codes only, written in subset of C language, i.e. for some construction it might return wrong bounds (e.g. for *switch* statement). Moreover, the runtime of *bounds* depends on Z3 library.

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 Trace Collector

Trace collector collects running times of C/C++ functions. The collected data are suitable for further postprocessing using the regression analysis and visualization by scatter plots.

Overview and Command Line Interface

perun collect trace

Generates *trace* 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
}
```

Trace 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.

The collector interface operates with two seemingly same concepts: (external) command and binary. External command refers to the script, executable, makefile, etc. that will be called / invoked during the profiling, such as 'make test', 'run_script.sh', './my_binary'. Binary, on the other hand, refers to the actual binary or executable file that will be profiled and contains specified functions / static probes etc. It is expected that the binary will be invoked / called as part of the external command script or that external command and binary are the same.

The interface for rules (functions, static probes) specification offers a way to specify profiled locations both with sampling or without it. Note that sampling can reduce the overhead imposed by the profiling. Static rules can be further paired - paired rules act as a start and end point for time measurement. Without a pair, the rule measures time between each two probe hits. The pairing is done automatically for static locations with convention <name> and <name>_end or <name>_END. Otherwise, it is possible to pair rules by the delimiter '#', such as <name1>#<name2>.

Trace 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 trace of individual functions.

Scatter plots are suitable visualization for profiles collected by *trace* 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 *Trace Collector* for more thorough description and examples of *trace* collector.

```
perun collect trace [OPTIONS]
```

Options

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

-f, --func <func>
Set the probe point for the given function.

-s, --static <static>
Set the probe point for the given static location.

-d, --dynamic <dynamic>
Set the probe point for the given dynamic location.

-fs, --func-sampled <func_sampled>
Set the probe point and sampling for the given function.

-ss, --static-sampled <static_sampled>
Set the probe point and sampling for the given static location.

-ds, --dynamic-sampled <dynamic_sampled>
Set the probe point and sampling for the given dynamic location.

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

--with-static, --no-static
The selected method will also extract and profile static probes.

-b, --binary <binary>
The profiled executable. If not set, then the command is considered to be the profiled executable and is used as a binary parameter

-t, --timeout <timeout>
Set time limit (in seconds) for the profiled command, i.e. the command will be terminated after reaching the time limit. Useful for endless commands etc.

-z, --zip-temps
Zip and compress the temporary files (SystemTap log, raw performance data, watchdog log ...) in the perun log directory before deleting them.

-k, --keep-temps
Do not delete the temporary files in the file system.

-vt, --verbose-trace
Set the trace file output to be more verbose, useful for debugging.

-q, --quiet
Reduces the verbosity of the collector info messages.

-w, --watchdog
Enable detailed logging of the whole collection process.

-o, --output-handling <output_handling>
Sets the output handling of the profiled command: - default: the output is displayed in the terminal - capture: the output is being captured into a file as well as displayed in the terminal (note that buffering causes a delay in the terminal output) - suppress: redirects the output to the DEVNULL

-i, --diagnostics
Enable detailed surveillance mode of the collector. The collector turns on detailed logging (watchdog), verbose trace, capturing output etc. and stores the logs and files in an archive (zip-temps) in order to provide as much diagnostic data as possible for further inspection.

Examples

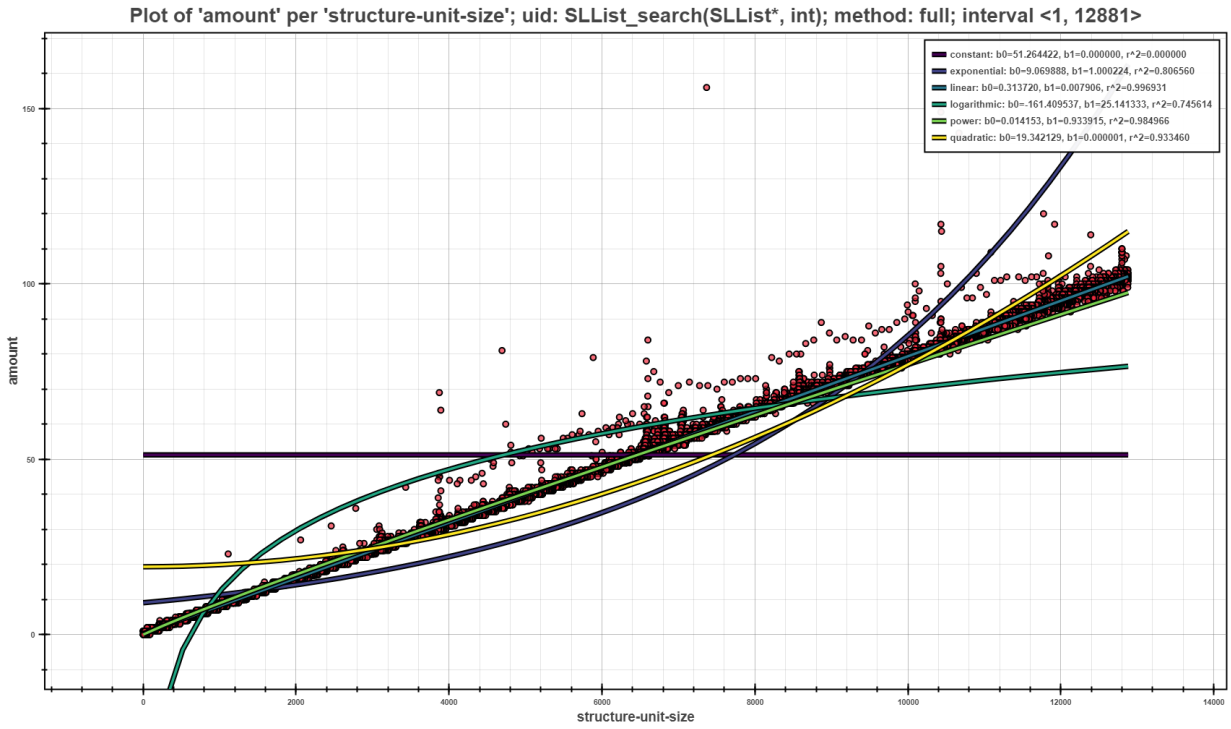

```
{
  "resources": {
    "SLList_insert(SLList*, int)#0": {
      "amount": [
        1, 0, 1, 1
      ],
      "structure-unit-size": [
        0, 1, 2, 3
      ]
    },
    "SLList_destroy(SLList*)#0": {
      "amount": [
        1
      ],
      "structure-unit-size": [
        4
      ]
    },
    "SLList_init(SLList*)#0": {
      "amount": [
        6
      ],
      "structure-unit-size": [
        0
      ]
    },
    "SLList_search(SLList*, int)#0": {
      "amount": [
        0
      ],
      "structure-unit-size": [
        0
      ]
    }
  },
  "header": {
    "workload": "",
```

```
    "type": "mixed",
    "units": {
      "mixed(time delta)": "us"
    },
    "params": "",
    "cmd": "../stap-collector/tst"
  },
  "models": [],
  "collector_info": {
    "params": {
      "global_sampling": null,
      "sampling": [
        {
          "func": "SLList_insert",
          "sample": 1
        },
        {
          "func": "func1",
          "sample": 1
        }
      ],
      "rules": [
        "SLList_init",
        "SLList_insert",
        "SLList_search",
        "SLList_destroy"
      ],
      "method": "custom"
    },
    "name": "complexity"
  },
  "resource_type_map": {
    "SLList_insert(SLList*, int)#0": {
      "subtype": "time delta",
      "uid": "SLList_insert(SLList*, int)",
      "time": "6.8e-05s",
      "type": "mixed"
    },
    "SLList_destroy(SLList*)#0": {
      "subtype": "time delta",
      "uid": "SLList_destroy(SLList*)",
      "time": "6.8e-05s",
      "type": "mixed"
    },
    "SLList_init(SLList*)#0": {
      "subtype": "time delta",
      "uid": "SLList_init(SLList*)",
      "time": "6.8e-05s",
      "type": "mixed"
    },
    "SLList_search(SLList*, int)#0": {
      "subtype": "time delta",
      "uid": "SLList_search(SLList*, int)",
      "time": "6.8e-05s",
      "type": "mixed"
    }
  },
  "postprocessors": [],
```

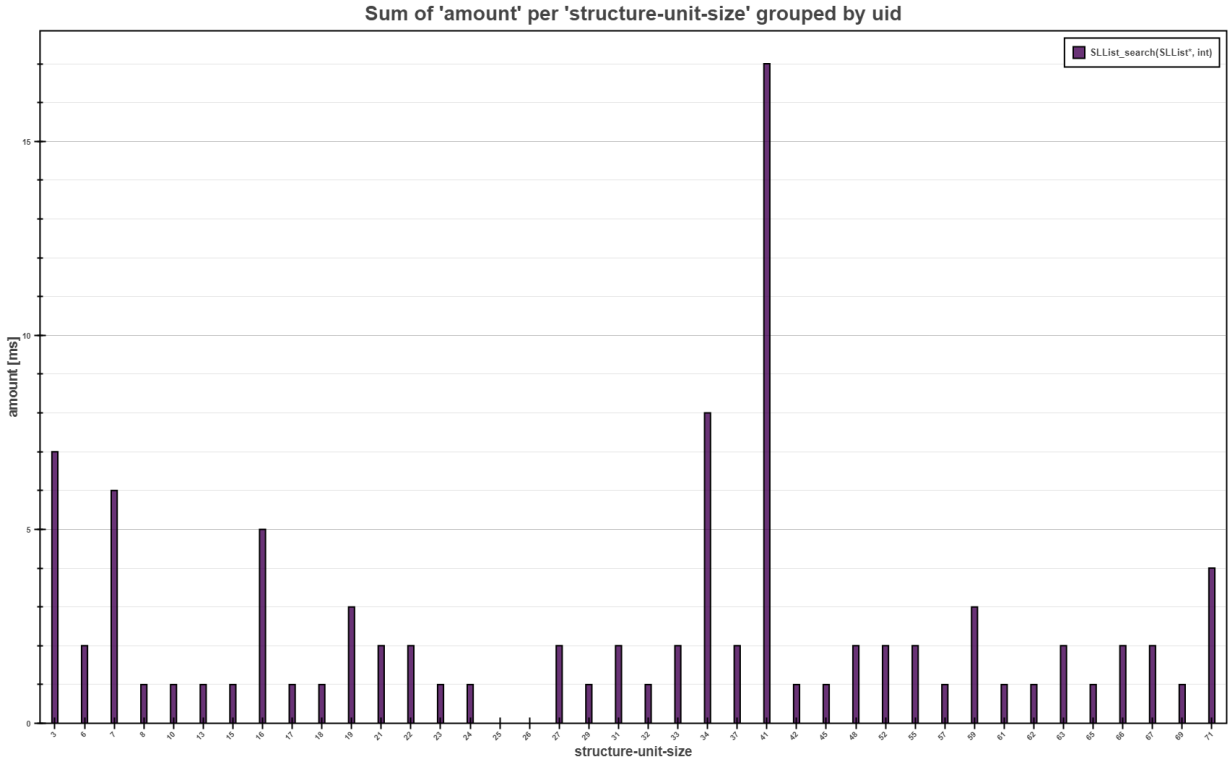
```
  "origin": "f7f3dcea69b97f2b03c421a223a770917149cfae"
}
```

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).

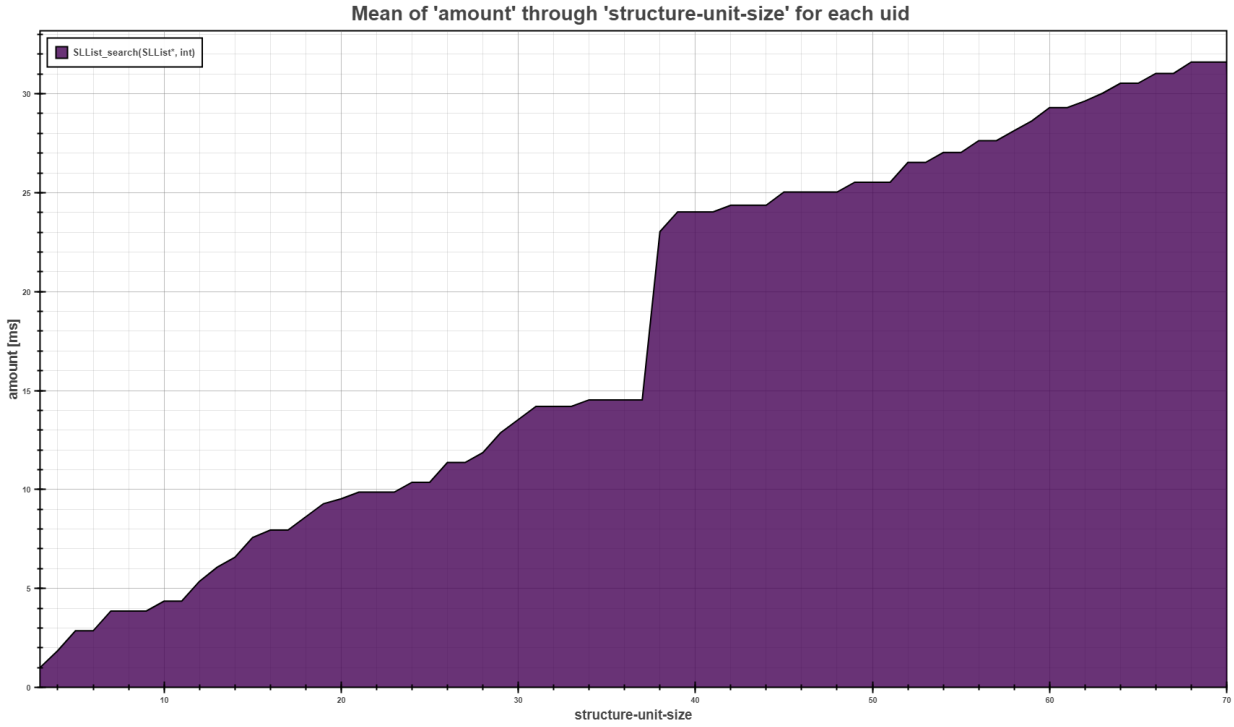
Highlighted lines show important keys and regions in the profile, e.g. the *origin*, `collector-info` or `resources`.



The *Scatter Plot* above shows the example of visualization of trace 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 profile, 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.

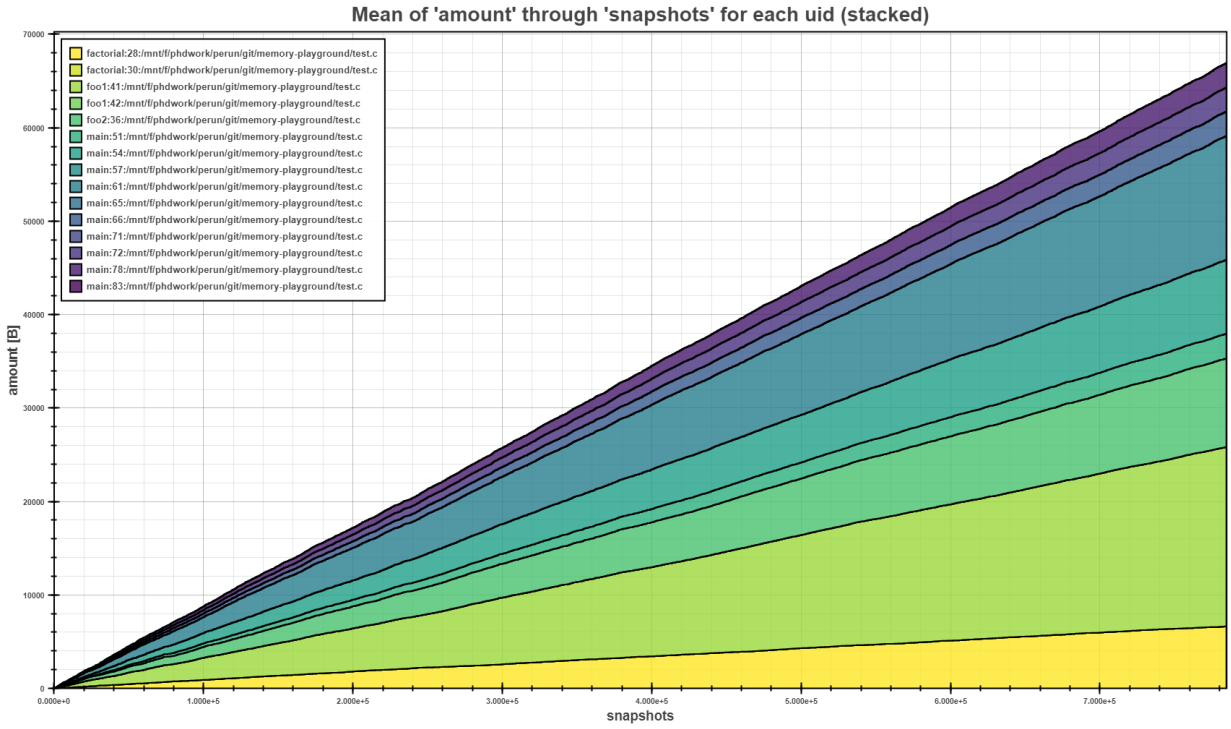
Examples


```
{
  "resources": {
    "../memory_collect_test.c:main:22#0": {
      "amount": [
        4
      ],
      "address": [
        19284560
      ]
    },
    "../memory_collect_test.c:main:27#0": {
      "amount": [
        0
      ],
      "address": [
        19284560
      ]
    }
  },
  "header": {
    "units": {
      "memory": "B"
    },
    "cmd": "./mct",
    "workload": "",
    "params": "",
    "type": "memory"
  },
  "models": [],
  "collector_info": {
    "params": {
      "all": false,
      "sampling": 0.025,
      "no_func": null,
      "no_source": null
    },
    "name": "memory"
  },
  "resource_type_map": {
    "../memory_collect_test.c:main:22#0": {
```

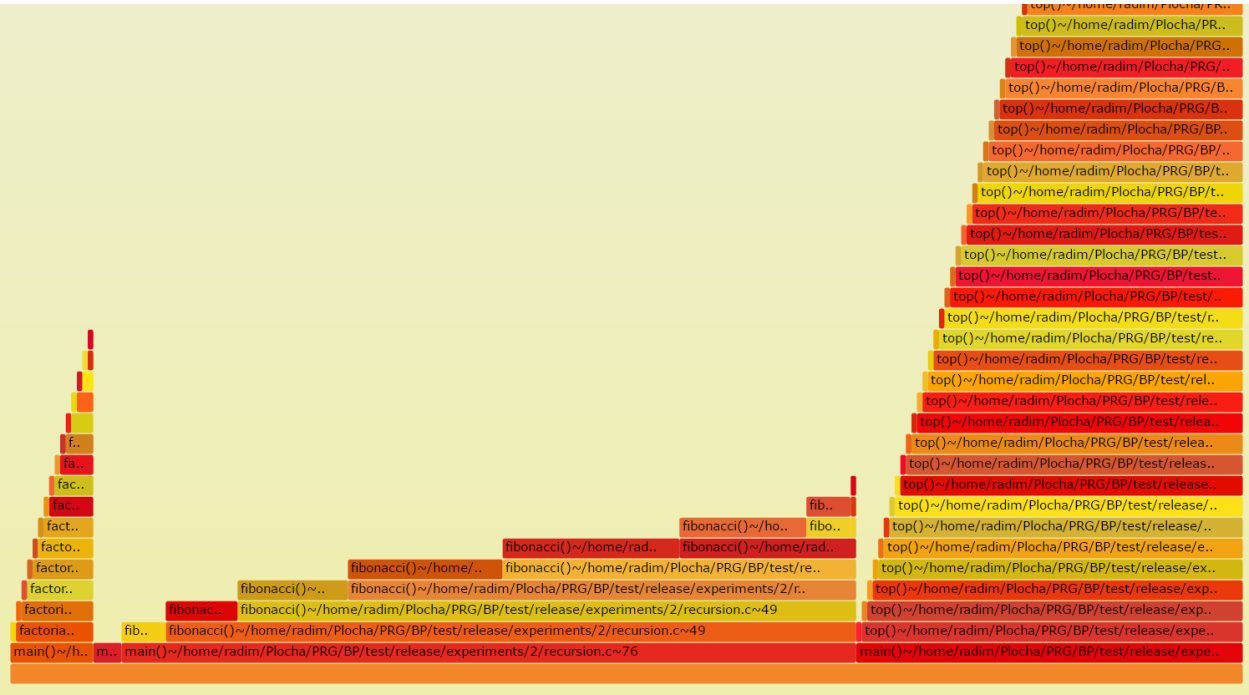
```
      "snapshot": 0,
      "time": "0.025000",
      "type": "memory",
      "trace": [
        {
          "function": "malloc",
          "line": 0,
          "source": "unreachable"
        },
        {
          "function": "main",
          "line": 22,
          "source": "../memory_collect_test.c"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

```
      "subtype": "free",
      "uid": {
        "function": "main",
        "line": 27,
        "source": "../memory_collect_test.c"
      }
    }
  },
  "postprocessors": [],
  "origin": "74288675e4074f1ad5bbb0d3b3253911ab42267a"
}
```

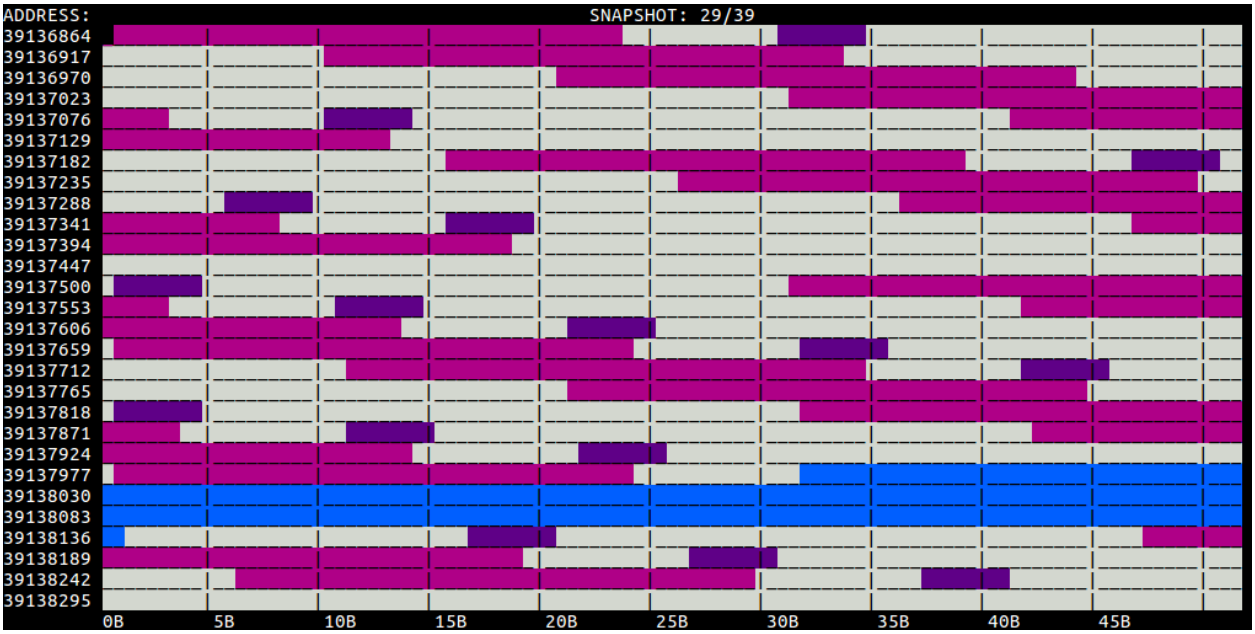

The above is an example of profiled data on a simple binary, which makes several minor allocations. Profile shows a simple allocation followed by deallocation and highlights important keys and regions in the *memory* profiles, e.g. the *origin*, `collector-info` or `resources`



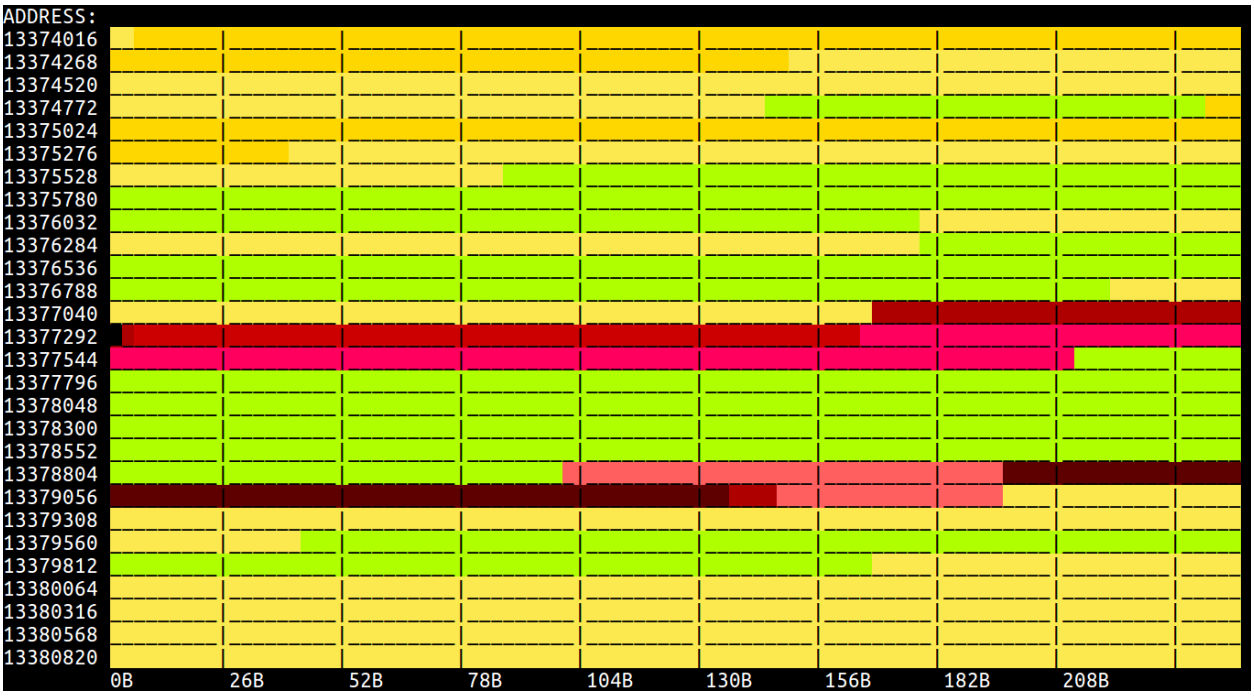
The *Flow Plot* above shows the mean of allocated amounts per each allocation site (i.e. `uid`) in stacked mode. The stacking of the means clearly shows, where the biggest allocations where made during the program run.



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",
    "subtype": "sys",
    "uid": cmd
    "order": 1
}
```

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

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

Options

-w, --warmup <warmup>
Before the actual timing, the collector will execute <int> warm-up executions.

-r, --repeat <repeat>
The timing of the given binaries will be repeated <int> times.

Examples


```
{
  "origin": "8de6cd99e4dc36cd73a2af906cde12456e96d9f1",
  "header": {
    "type": "time",
    "params": "",
    "units": {
      "time": "s"
    },
    "cmd": "./list_search",
    "workload": "100000"
  },
  "collector_info": {
    "params": {
      "repeat": 2,
      "warmup": 3
    },
    "name": "time"
  },
  "postprocessors": [],
  "global": {
    "timestamp": 0.565476655960083,
    "resources": [
      {
        "subtype": "real",
        "uid": "./list_search",
        "order": 1,
        "type": "time",
        "amount": 0.26
      },
      {
        "subtype": "user",
        "uid": "./list_search",
        "order": 1,
        "type": "time",
        "amount": 0.25
      },
      {
        "subtype": "sys",
        "uid": "./list_search",
        "order": 1,
        "type": "time",
        "amount": 0.0
```

```
      },
      {
        "subtype": "real",
        "uid": "./list_search",
        "order": 2,
        "type": "time",
        "amount": 0.27
      },
      {
        "subtype": "user",
        "uid": "./list_search",
        "order": 2,
        "type": "time",
        "amount": 0.28
      },
      {
        "subtype": "sys",
        "uid": "./list_search",
        "order": 2,
        "type": "time",
        "amount": 0.0
      }
    ]
  },
}
```


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

4.1.4 Bounds Collector

Automatic analysis of resource bounds of C programs.

Bounds collector employs a technique of _loopus tool, which performs an amortized analysis of input C program. Loopus is limited to integer programs only, and for each function and for each loop it computes a symbolic bound (e.g. 2*n + max(0, m)). Moreover, it computes the big-O notation highlighting the main source of the complexity.

Overview and Command Line Interface

perun collect bounds

Generates *memory* performance profile, 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.

```
{
        "uid": {
            "source": "../test.c",
            "function": "main",
            "line": 22
            "column": 40
        }
        "bound": "1 + max(0, (k + -1))",
        "class": "O(n^1)"
        "type": "bound",
}
```

```
`Memory` profiles can be efficiently interpreted using :ref:`views-heapmap`
technique (together with its `heat` mode), which shows memory allocations
(by functions) in memory address map.

Refer to :ref:`collectors-memory` for more thorough description and
examples of `memory` collector.
```

```
perun collect bounds [OPTIONS]
```

Options

-s, --source, --src <source>
: Source C file that will be analyzed.

-d, --source-dir <source_dir>
: Directory, where source C files are stored. All of the existing files with valid extensions (.c).

Examples


```
{
  "origin": "409dd7468a328038c9a1ea5a6a0f7baa89f8997a",
  "header": {
    "type": "bound",
    "args": "",
    "workload": "",
    "units": {
      "bound": "iterations"
    },
    "cmd": "partitioning"
  },
  "resources": {
    "int_partitioning.c:partitioning:34:42#0": {},
    "int_partitioning.c:partitioning:49:60#0": {},
    "int_partitioning.c:partitioning:64:69#0": {},
    "int_partitioning.c:partitioning:18:4#0": {},
    "int_partitioning.c:partitioning:55:58#0": {}
  },
  "models": [],
  "postprocessors": [],
  "collector_info": {
    "params": {
```

```
      "sources": [
        "int_partitioning.c"
      ],
      "workload": "",
      "source_dir": [],
      "source": [
        "int_partitioning.c"
      ]
    },
    "name": "bounds"
  },
  "resource_type_map": {
    "int_partitioning.c:partitioning:34:42#0": {
      "uid": {
        "line": 34,
        "column": 42,
        "source": "int_partitioning.c",
        "function": "partitioning"
      },
      "type": "local bound",
      "bound": "1 + max(0, (k + -1))",
      "time": "0.0",
      "class": "O(n^1)"
    },
    "int_partitioning.c:partitioning:49:60#0": {
      "uid": {
        "line": 49,
        "column": 60,
        "source": "int_partitioning.c",
        "function": "partitioning"
      },
      "type": "local bound",
      "bound": "2 + max(0, (k + -1))",
      "time": "0.0",
      "class": "O(n^1)"
    },
    "int_partitioning.c:partitioning:64:69#0": {
      "uid": {
        "line": 64,
        "column": 69,
        "source": "int_partitioning.c",
        "function": "partitioning"
      },
      "type": "local bound",
      "bound": "2 + max(0, (k + -1))",
      "time": "0.0",
      "class": "O(n^1)"
    },
    "int_partitioning.c:partitioning:18:4#0": {
      "uid": {
        "line": 18,
        "column": 4,
        "source": "int_partitioning.c",
        "function": "partitioning"
      },
      "type": "total bound",
      "bound": "6 + 4 × max(0, (k + -1))",
      "time": "0.0",
```

```
      "class": "O(n^1)"
    },
    "int_partitioning.c:partitioning:55:58#0": {
      "uid": {
        "line": 55,
        "column": 58,
        "source": "int_partitioning.c",
        "function": "partitioning"
      },
      "type": "local bound",
      "bound": "1 + max(0, (k + -1))",
      "time": "0.0",
      "class": "O(n^1)"
    }
  }
}
```

The above is an example of profiled data using the *bounds* with important regions and keys highlighted. The bounds corresponds to the program listed below, which contains four highlighted loops. For each loop we have a local bound that is represented as a ranking function based on input function parameters. For each bound, we also list its class, i.e. the highest polynom of the ranking function, or Big-O complexity. In case, the complexity cannot be inferred, the Loopus returns failure and we report infinite bound, which is safe approximation. Each function then gets a cummulative total bound, that represents the whole complexity of the function.

```
int partitioning(unsigned int k) {
    TList *list, *temp;

    unsigned int list_next_NULL;
    unsigned int list_next_p;
    unsigned int p_next_NULL;
    unsigned int x_next_NULL;
    unsigned int y_next_x;
    list = malloc(sizeof(TList));
    list->next = NULL;

    // Create nondeterminstic list
    TList *p = list;
    list_next_NULL = 1;
    list_next_p = 0;
    while(k > 1) {
        temp = malloc(sizeof(TList));
        temp->next = NULL;
        p->next = temp;
        p = temp;
        list_next_NULL = list_next_p + 2;
        list_next_p += 1;
        --k;
    }

    // Traverse the list
    TList* x = list;
    TList* y = x;
    x_next_NULL = list_next_NULL;
    y_next_x = 0;
    while(x_next_NULL > 0 && x != NULL) {
        x = x->next;
        x_next_NULL -= 1;
```

```
        y_next_x += 1;
        // The end will always jump out
        if(NONDET) {
            while(y_next_x > 0 && y != x) {
                y = y->next;
                y_next_x -= 1;
            }
        }
    }

    p = list;
    p_next_NULL = list_next_NULL;
    while(p_next_NULL > 0 && p != NULL) {
        temp = p;
        p = p->next;
        free(temp);
        p_next_NULL -= 1;
    }

    return 0;
}

```

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
    |-- /trace
    |-- /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/e/phdwork/perun/git/docs/_static/templates/collectors_run.py
+++ /mnt/e/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/e/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/e/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪collectors.py
@@ -6,7 +6,7 @@
         ))
     return {
         'vcs': ['git'],
-        'collect': ['trace', 'memory', 'time'],
+        'collect': ['trace', 'memory', 'time', 'mycollector'],
         'postprocess': ['filter', 'normalizer', 'regression-analysis'],
         '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 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:

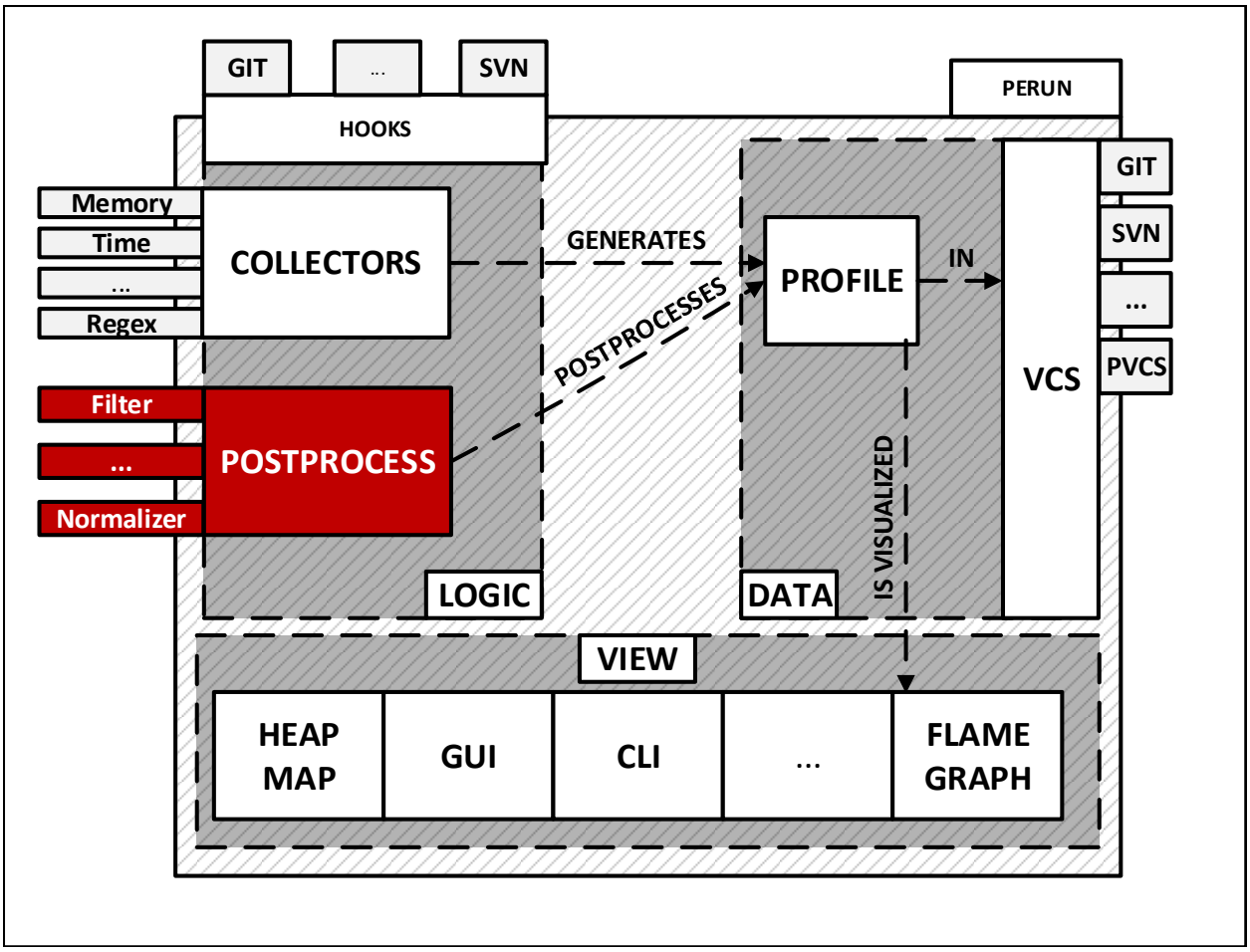
1. By **Directly running postprocessors** through `perun postprocessby` command, that takes the profile (either stored or pending) and uses a single postprocessor with given configuration.
2. By **Using job specification** either as a single run of batch of profiling jobs using `perun run job` or according 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 five 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.
3. *Clusterizer* tries to classify resources to uniquely identified clusters, which can be used for further postprocessing (e.g. by regression analysis) or to group similar amounts of resources.
4. *Regressogram method* (authored by **Simon Stupinsky**) also known as the binning approach, is the simplest non-parametric estimator. This method trying to fit models through data by dividing the interval into N equal-width bucket and the resultant value in each bucket is equal to result of selected statistical aggregation function (mean/median) within the values in the relevant bucket. In short, we can describe the regressogram as a step function (i.e. constant function by parts).
5. *Moving Average Methods* (authored by **Simon Stupinsky**) also know as the rolling average or running average, is the statistical analysis belongs to non-parametric approaches. This method is based on the analysis of the

given data points by creating a series of values based on the specific aggregation function, most often average or possibly median. The resulting values are derived from the different subsets of the full data set. We currently support the two main methods of this approach and that the **Simple** Moving Average and the **Exponential** Moving Average. In the first method is an available selection from two aggregation function: **mean** or **median**.

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 *Creating your own Postprocessor*.

5.1.1 Normalizer Postprocessor

Normalizer is a simple postprocessor that normalizes the values.

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**: *Trace 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_start": 0,
    "x_end": 11892,
    "model": "linear",
    "method": "full",
}
```

Note that if your data are not suitable for regression analysis, check out *Clusterizer* to postprocess your profile to be analysable by this analysis.

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

-dp, --depending-on <per_key>
Sets the key that will be used as a source of independent variable.

-o, --of <of_key>
Sets key for which we are finding the model.

Examples

```
{
  "resources": {
    "SLList_insert(SLList*, int)#0": {
      "amount": [
        1, 0, 1, 1
      ],
      "structure-unit-size": [
        0, 1, 2, 3
      ]
    },
    "SLList_destroy(SLList*)#0": {
      "amount": [
        1
      ],
      "structure-unit-size": [
        4
      ]
    },
    "SLList_init(SLList*)#0": {
      "amount": [
```

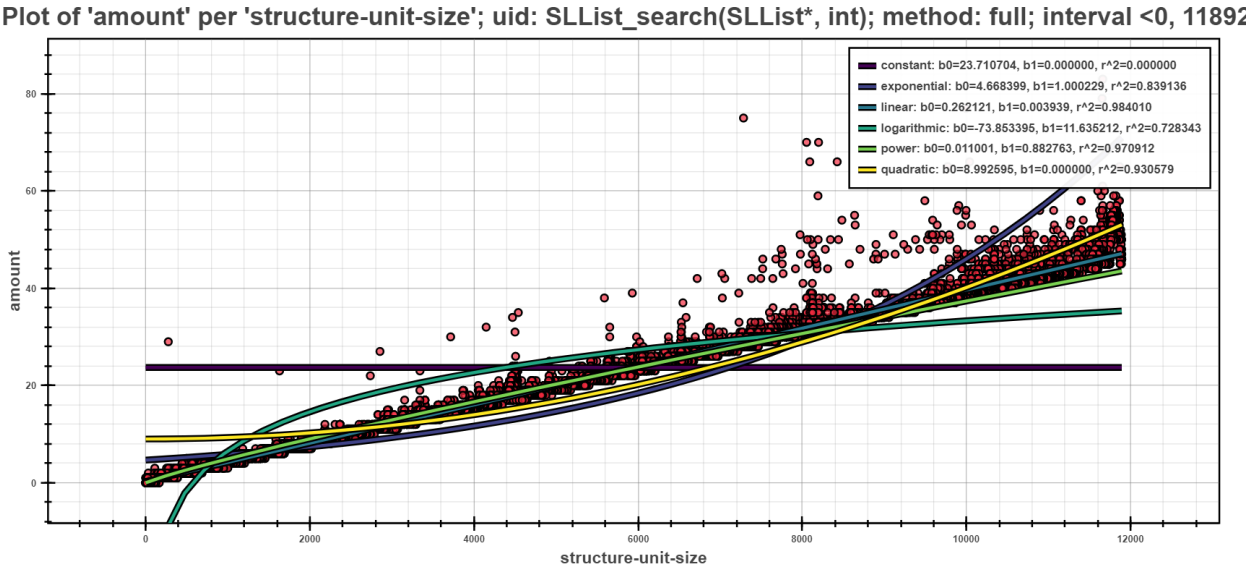
```
        6
      ],
      "structure-unit-size": [
        0
      ]
    },
    "SLList_search(SLList*, int)#0": {
      "amount": [
        0
      ],
      "structure-unit-size": [
        0
      ]
    }
  },
  "header": {
    "workload": "",
    "type": "mixed",
    "units": {
      "mixed(time delta)": "us"
    },
    "params": "",
    "cmd": "../stap-collector/tst"
  },
  "models": [
    {
      "coeffs": [
        {
          "value": 0.75,
          "name": "b0"
        },
        {
          "value": 0.0,
          "name": "b1"
        }
      ],
      "method": "full",
      "r_square": 0.0,
      "model": "constant",
      "uid": "SLList_insert(SLList*, int)",
      "x_interval_end": 3,
      "x_interval_start": 0
    },
    {
      "coeffs": [
        {
          "value": 1.0,
          "name": "b0"
        },
        {
          "value": 1.0,
          "name": "b1"
        }
      ],
      "method": "full",
      "r_square": 0.0,
      "model": "exponential",
      "uid": "SLList_insert(SLList*, int)",
```

```
      "x_interval_end": 3,
      "x_interval_start": 0
    },
    {
      "coeffs": [
        {
          "value": 0.6,
          "name": "b0"
        },
        {
          "value": 0.1,
          "name": "b1"
        }
      ],
      "method": "full",
      "r_square": 0.06666666666666667,
      "model": "linear",
      "uid": "SLList_insert(SLList*, int)",
      "x_interval_end": 3,
      "x_interval_start": 0
    },
    {
      "coeffs": [
        {
          "value": 0.08877935258260898,
          "name": "b0"
        },
        {
          "value": 0.9675751528184126,
          "name": "b1"
        }
      ],
      "method": "full",
      "r_square": 0.8668309711260865,
      "model": "logarithmic",
      "uid": "SLList_insert(SLList*, int)",
      "x_interval_end": 3,
      "x_interval_start": 0
    },
    {
      "coeffs": [
        {
          "value": 1.0,
          "name": "b0"
        },
        {
          "value": 0.0,
          "name": "b1"
        }
      ],
      "method": "full",
      "r_square": 0.0,
      "model": "power",
      "uid": "SLList_insert(SLList*, int)",
      "x_interval_end": 3,
      "x_interval_start": 0
    },
    {
```

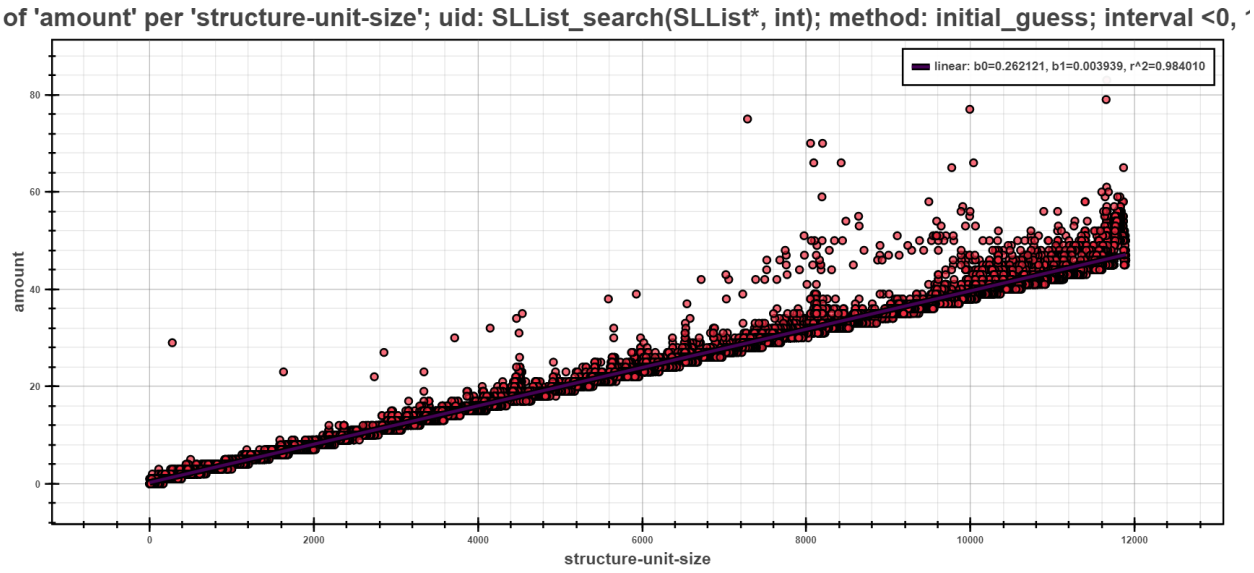
```
      "coeffs": [
        {
          "value": 0.5714285714285714,
          "name": "b0"
        },
        {
          "value": 0.05102040816326531,
          "name": "b1"
        }
      ],
      "method": "full",
      "r_square": 0.17006802721088435,
      "model": "quadratic",
      "uid": "SLList_insert(SLList*, int)",
      "x_interval_end": 3,
      "x_interval_start": 0
    }
  ],
  "collector_info": {
    "params": {
      "global_sampling": null,
      "sampling": [
        {
          "func": "SLList_insert",
          "sample": 1
        },
        {
          "func": "func1",
          "sample": 1
        }
      ],
      "rules": [
        "SLList_init",
        "SLList_insert",
        "SLList_search",
        "SLList_destroy"
      ],
      "method": "custom"
    },
    "name": "complexity"
  },
  "resource_type_map": {
    "SLList_insert(SLList*, int)#0": {
      "subtype": "time delta",
      "uid": "SLList_insert(SLList*, int)",
      "time": "6.8e-05s",
      "type": "mixed"
    },
    "SLList_destroy(SLList*)#0": {
      "subtype": "time delta",
      "uid": "SLList_destroy(SLList*)",
      "time": "6.8e-05s",
      "type": "mixed"
    },
    "SLList_init(SLList*)#0": {
      "subtype": "time delta",
      "uid": "SLList_init(SLList*)",
      "time": "6.8e-05s",
```

```
      "type": "mixed"
    },
    "SLList_search(SLList*, int)#0": {
      "subtype": "time delta",
      "uid": "SLList_search(SLList*, int)",
      "time": "6.8e-05s",
      "type": "mixed"
    }
  },
  "postprocessors": [],
  "origin": "f7f3dcea69b97f2b03c421a223a770917149cfae"
}
```


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.

5.1.3 Clusterizer

A postprocessor that attempts to classify resources to clusters.

The main usage of this postprocessors is to prepare any kind of profile for further postprocessing, mainly by *Regression Analysis*. The clusterization is either realized w.r.t the sorted order of the resources or sliding window, with parametric width and height.

Command Line Interface

perun postprocessby clusterizer

Clusters each resource to an appropriate cluster in order to be postprocessable by regression analysis.

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

Clusterizer tries to find a suitable cluster for each resource in the profile. The clusters are either computed w.r.t the sort order of the resource amounts, or are computed according to the sliding window.

The sliding window can be further adjusted by setting its **width** (i.e. how many near values on the x axis will we fit to a cluster) and its **height** (i.e. how big of an interval of resource amounts will be consider for one cluster). Both **width** and **height** can be further augmented. **Width** can either be *absolute*, where we take in maximum the absolute number of resources, *relative*, where we take in maximum the percentage of number of resources for each cluster, or *weighted*,

where we take the number of resource depending on the frequency of their occurrences. Similarly, the **height** can either be *absolute*, where we set the interval of amounts to an absolute size, or *relative*, where we set the interval of amounts relative to the to the first resource amount in the cluster (so e.g. if we have window of height 0.1 and the first resource in the cluster has amount of 100, we will cluster every resources in interval 100 to 110 to this cluster).

For more details about regression analysis refer to *Clusterizer*.

```
perun postprocessby clusterizer [OPTIONS]
```

Options

-s, --strategy <strategy>
Specifies the clustering strategy, that will be applied for the profile

-wh, --window-height <window_height>
Specifies the height of the window (either fixed or proportional)

-rwh, --relative-window-height
Specifies that the height of the window is relative to the point

-fwh, --fixed-window-height
Specifies that the height of the window is absolute to the point

-ww, --window-width <window_width>
Specifies the width of the window, i.e. how many values will be taken by window.

-rww, --relative-window-width
Specifies whether the width of the window is weighted or fixed

-fww, --fixed-window-width
Specifies whether the width of the window is weighted or fixed

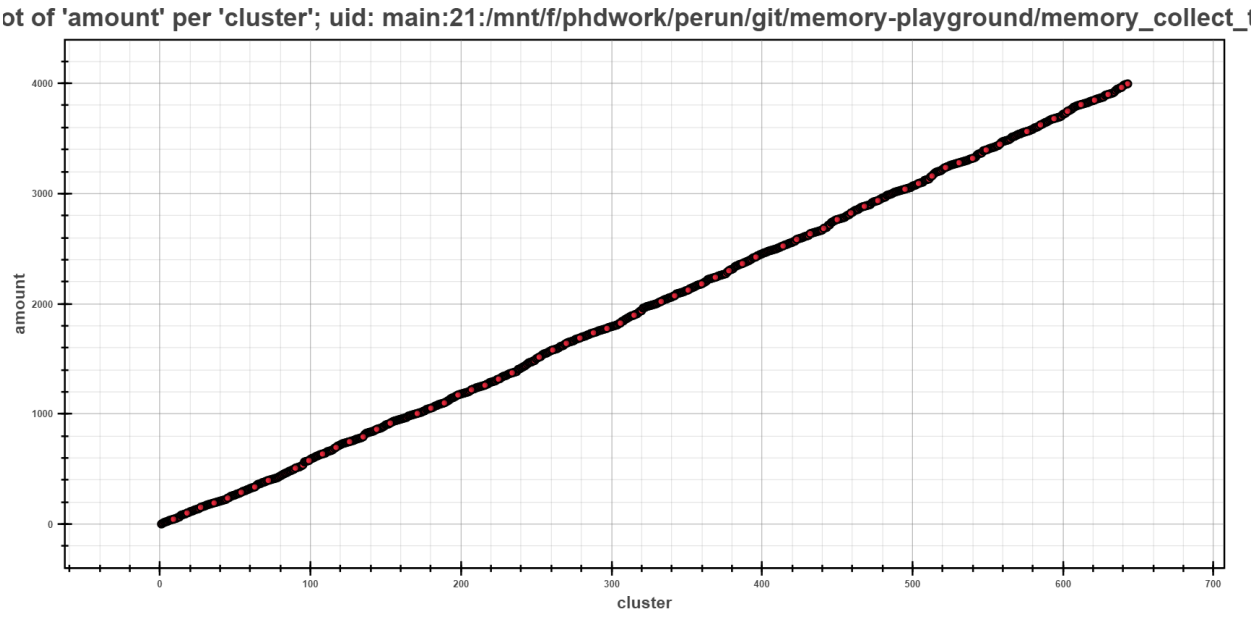
-www, --weighted-window-width
Specifies whether the width of the window is weighted or fixed

Examples


```
{
  "snapshots": [
    {
      "time": "5.000000",
      "resources": [
        {
          "amount": 0,
          "trace": [
            {
              "line": 0,
              "function": "malloc",
              "source": "unreachable"
            },
            {
              "line": 21,
              "function": "main",
              "source": "../memory_collect_test.c"
            },
            {
              "line": 0,
```

```
                "function": "__libc_start_main",
                "source": "unreachable"
              },
              {
                "line": 0,
                "function": "_start",
                "source": "unreachable"
              }
            ],
            "address": 31584848,
            "uid": "../memory_collect_test.c:main#22",
            "cluster": 1,
            "type": "memory",
            "subtype": "malloc"
          },
        ]
      }
  ]
}
```

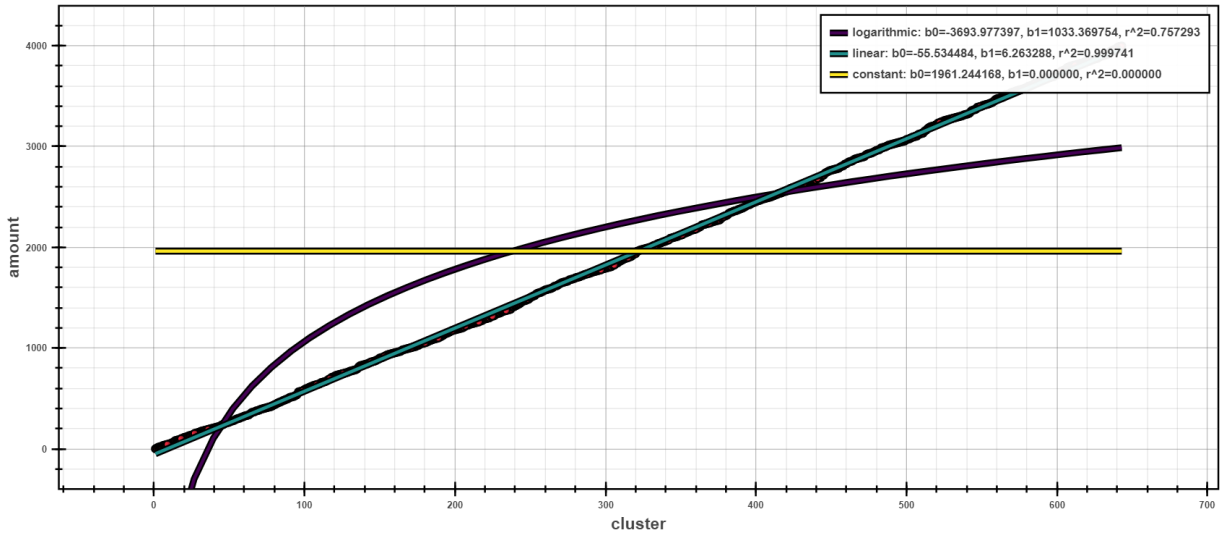

The profile above shows an example of profile postprocessed by clusterizer (note that this is only an excerpt of the whole profile). Each resource is annotated by a new field named `cluster`, which can be used in further interpretation of the profiles (either by *Bars Plot*, *Scatter Plot* or *Regression Analysis*).



The *Scatter Plot* above shows the memory profile of a simple example, which randomly allocates memory with linear dependency and was collected by *Memory Collector*. Since *Memory Collector* does not collect any other information, but memory rallocation records. Such profile cannot be used to infer any models. However the *Scatter Plot* above was postprocessed by clusterizer and hence, we can plot the dependency of amount of allocated memory per each cluster. The *Scatter Plot* itself ephasize the linear dependency of allocated memory depending on some unknown parameters (here represented by *cluster*).

We can use *Regression Analysis* to prove our assumption, and on the plot below we can see that the best model for the amount of allocated memory depending on clusters is indeed **linear**.

er 'cluster'; uid: 21:main:/mnt/f/phdwork/perun/git/memory-playground/memory_collect_test.c; method:



5.1.4 Regressogram method

Postprocessing of input profiles using the non-parametric method: regressogram. This method serves for finding fitting models for trends in the captured profiling resources using the constant function at the individual parts of the whole interval.

Command Line Interface

perun postprocessby regressogram

Execution of the interleaving of profiled resources by **regressogram** models.

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

Regressogram belongs to the simplest non-parametric methods and its properties are the following:

> **Regressogram**: can be described such as step function (i.e. constant function by parts). Regressogram uses the same basic idea as a histogram for density estimate. This idea is in dividing the set of values of the x-coordinates (<*per_key*>) into intervals and the estimate of the point in concrete interval takes the mean/median of the y-coordinates (<*of_resource_key*>), respectively of its value on this sub-interval. We currently use the *coefficient of determination* (R^2) to measure the fitness of regressogram. The fitness of estimation of regressogram model depends primarily on the number of buckets into which the interval will be divided. The user can choose number of buckets manually (<*bucket_window*>) or use one of the following methods to estimate the optimal number of buckets (<*bucket_method*>):
>
> - **sqrt**: square root (of data size) estimator, used for its speed and simplicity
> - **rice**: does not take variability into account, only data size and commonly overestimates
> - **scott**: takes into account data variability and data size, less robust estimator
> - **stone**: based on leave-one-out cross validation estimate of the integrated squared error
> - **fd**: robust, takes into account data variability and data size, resilient to outliers

- **sturges**: only accounts for data size, underestimates for large non-gaussian data
- **doane**: generalization of Sturges' formula, works better with non-gaussian data
- **auto**: max of the Sturges' and 'fd' estimators, provides good all around performance

For more details about these methods to estimate the optimal number of buckets or to view the code of these methods, you can visit SciPy.

For more details about this approach of non-parametric analysis refer to *Regressogram method*.

```
perun postprocessby regressogram [OPTIONS]
```

Options

-bn, --bucket_number <bucket_number>
Restricts the number of buckets to which will be placed the values of the selected statistics.

-bm, --bucket_method <bucket_method>
Specifies the method to estimate the optimal number of buckets.

-sf, --statistic_function <statistic_function>
Will use the <statistic_function> to compute the values for points within each bucket of regressogram.

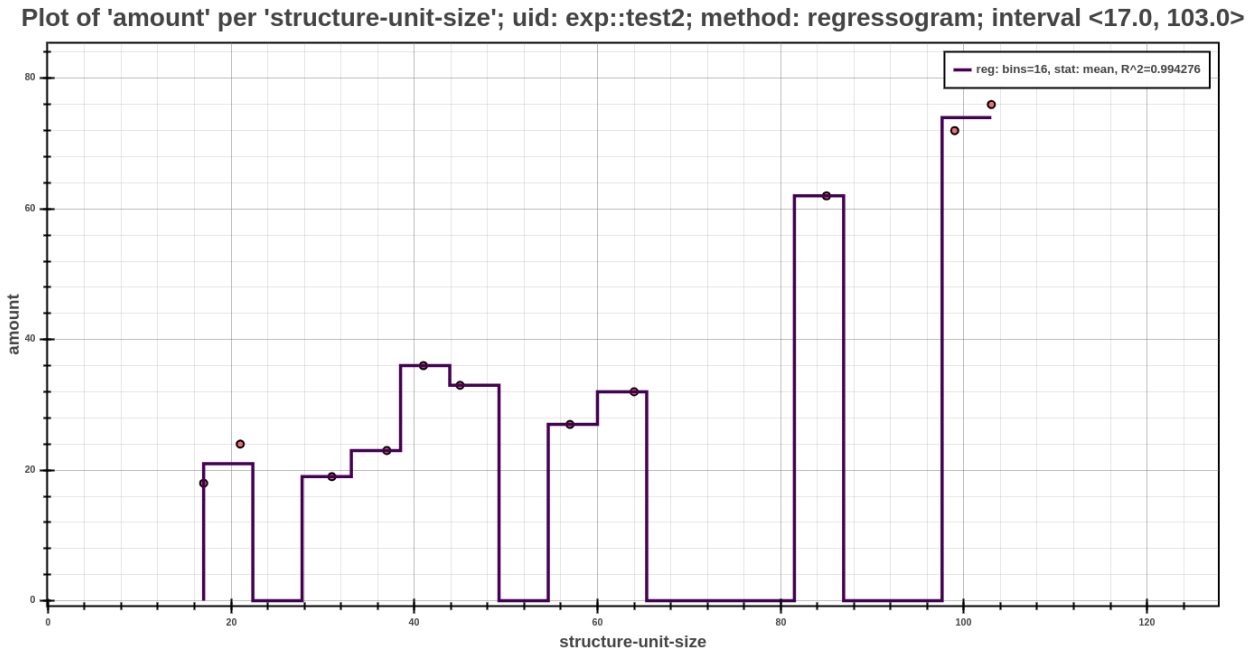
-of, --of-key <of_key>
Sets key for which we are finding the model (y-coordinates).

-per, --per-key <per_key>
Sets the key that will be used as a source variable (x-coordinates).

Examples


```
{
    "bucket_stats": [
        13.0,
        25.5
    ],
    "uid": "linear::test2",
    "bucket_method": "doane",
    "method": "regressogram",
    "r_square": 0.7575757575757576,
    "x_end": 9.0,
    "statistic_function": "mean",
    "x_start": 0.0
}
```


The example above shows an example of profile post-processed by regressogram method (note that this is only an excerpt of the whole profile). Each such model of shows the computed values in the individual buckets, that are represented by *bucket_stats*. The next value in this example is *statistic_function*, which represented the statistic to compute the value in each bucket. Further contains the name of the method (*bucket_method*) by which was calculated the optimal number of buckets, in this case specifically computed with Doanes formula, and *coefficient of determination* (R^2) for measuring the fitting of the model. Each such model can be used in the further interpretation of the models (either by *Scatter Plot* or *Average Amount Threshold*).



The *Scatter Plot* above shows the interpreted model, computed using the **regressogram** method. In the picture, one can see that the dependency of running time based on the structural size is best fitted by *exponential* models.

5.1.5 Moving Average Methods

Postprocessing of input profiles using the non-parametric method: moving average. This method serves to analyze data points in the captured profiling resources by creating a series of averages, eventually medians, of different subsets of the full data set.

Command Line Interface

perun postprocessby moving_average

Execution of the interleaving of profiled resources by *moving average* models.

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

Moving average methods are the natural generalizations of regressogram method. This method uses the local averages/medians of y-coordinates (<*of_resource_key*>), but the estimate in the x-point (<*per_key*>) is based on the centered surroundings of this points, more precisely:

> **Moving Average**: is a widely used estimator in the technical analysis, that helps smooth the dataset by filtering out the 'noise'. Among the basic properties of this methods belongs the ability to reduce the effect of temporary variations in data, better improvement of the fitness of data to a line, so called smoothing, to show the data's trend more clearly and highlight any value below or above the trend. The most important task with this type of non-parametric approach is the choice of the <*window-width*>. If the user does not choose it, we try approximate this value by using the value of *coefficient of determination* (R^2). At the begin of the analysis is set the initial value of window width and then follows the interleaving of

the current dataset, which runs until the value of *coefficient of determination* will not reach the required level. By this way is guaranteed the desired smoothness of the resulting models. The two basic and commonly used <*moving-methods*> are the **simple** moving average (**sma**) and the *exponential* moving average (**ema**).

For more details about this approach of non-parametric analysis refer to *Moving Average Methods*.

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

Options

-mp, --min_periods <min_periods>
Provides the minimum number of observations in window required to have a value. If the number of possible observations smaller then result is NaN.

-of, --of-key <of_key>
Sets key for which we are finding the model (y-coordinates).

-per, --per-key <per_key>
Sets the key that will be used as a source variable (x-coordinates).

Commands

ema

sma

smm

perun postprocessby moving_average sma

Simple Moving Average

In the most of cases, it is an unweighted Moving Average, this means that the each x-coordinate in the data set (profiled resources) has equal importance and is weighted equally. Then the *mean* is computed from the previous *n data* (<*no-center*>), where the *n* marks <*window-width*>. However, in science and engineering the mean is normally taken from an equal number of data on either side of a central value (<*center*>). This ensures that variations in the mean are aligned with the variations in the mean are aligned with variations in the data rather than being shifted in the x-axis direction. Since the window at the boundaries of the interval does not contain enough count of points usually, it is necessary to specify the value of <*min-periods*> to avoid the NaN result. The role of the weighted function in this approach belongs to <*window-type*>, which represents the suite of the following window functions for filtering:

- **boxcar**: known as rectangular or Dirichlet window, is equivalent to no window at all: –
- **triang**: standard triangular window
- **blackman**: formed by using three terms of a summation of cosines, minimal leakage, close to optimal
- **hamming**: formed by using a raised cosine with non-zero endpoints, minimize the nearest side lobe
- **bartlett**: similar to triangular, endpoints are at zero, processing of tapering data sets
- **parzen**: can be regarded as a generalization of k-nearest neighbor techniques
- **bohman**: convolution of two half-duration cosine lobes
- **blackmanharris**: minimum in the sense that its maximum side lobes are minimized (symmetric 4-term)
- **nuttall**: minimum 4-term Blackman-Harris window according to Nuttall (so called 'Nuttall4c')
- **barthann**: has a main lobe at the origin and asymptotically decaying side lobes on both sides

- **kaiser**: formed by using a Bessel function, needs beta value (set to 14 - good starting point)

For more details about this window functions or for their visual view you can see SciPyWindow.

```
perun postprocessby moving_average sma [OPTIONS]
```

Options

-wt, --window_type <window_type>
Provides the window type, if not set then all points are evenly weighted. For further information about window types see the notes in the documentation.

--center, --no-center
If set to False, the result is set to the right edge of the window, else is result set to the center of the window

-ww, --window_width <window_width>
Size of the moving window. This is a number of observations used for calculating the statistic. Each window will be a fixed size.

perun postprocessby moving_average smm

Simple Moving Median

The second representative of Simple Moving Average methods is the Simple Moving **Median**. For this method are applicable to the same rules like in the first described method, except for the option for choosing the window type, which do not make sense in this approach. The only difference between these two methods are the way of computation the values in the individual sub-intervals. Simple Moving **Median** is not based on the computation of average, but as the name suggests, it based on the **median**.

```
perun postprocessby moving_average smm [OPTIONS]
```

Options

--center, --no-center
If set to False, the result is set to the right edge of the window, else is result set to the center of the window

-ww, --window_width <window_width>
Size of the moving window. This is a number of observations used for calculating the statistic. Each window will be a fixed size.

perun postprocessby moving_average ema

Exponential Moving Average

This method is a type of moving average methods, also know as **Exponential** Weighted Moving Average, that places a greater weight and significance on the most recent data points. The weighting for each far x-coordinate decreases exponentially and never reaching zero. This approach of moving average reacts more significantly to recent changes than a *Simple* Moving Average, which applies an equal weight to all observations in the period. To calculate an EMA must be first computing the **Simple** Moving Average (SMA) over a particular sub-interval. In the next step must be calculated the multiplier for smoothing (weighting) the EMA, which depends on the selected formula, the following options are supported (<*decay*>):

- **com**: specify decay in terms of center of mass: α = 1 / (1 + com), for com >= 0

- **span**: specify decay in terms of span: α = 2 / (span + 1), for span >= 1
- **halflife**: specify decay in terms of half-life, α = 1 - exp(log(0.5) / halflife), for halflife > 0
- **alpha**: specify smoothing factor α directly: 0 < α <= 1

The computed coefficient α represents the degree of weighting decrease, a constant smoothing factor, The higher value of α discounts older observations faster, the small value to the contrary. Finally, to calculate the current value of EMA is used the relevant formula. It is important do not confuse **Exponential** Moving Average with **Simple** Moving Average. An **Exponential** Moving Average behaves quite differently from the second mentioned method, because it is the function of weighting factor or length of the average.

```
perun postprocessby moving_average ema [OPTIONS]
```

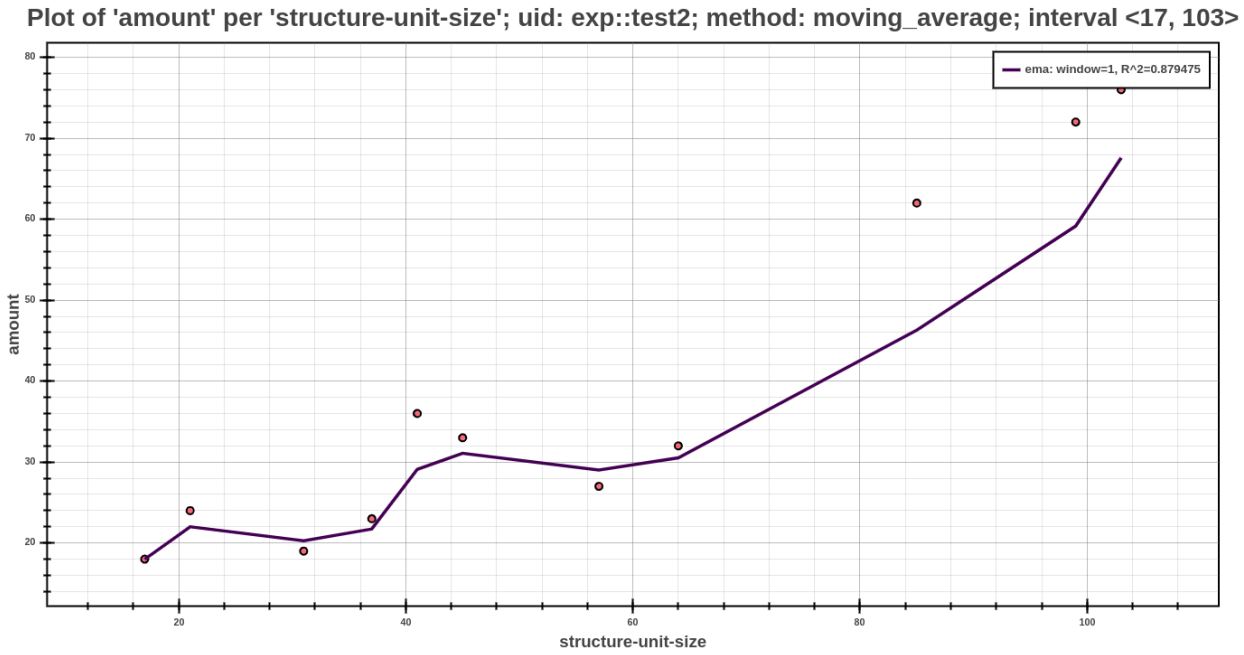
Options

-d, --decay <decay>
Exactly one of "com", "span", "halflife", "alpha" can be provided. Allowed values and relationship between the parameters are specified in the documentation (e.g. –decay=com 3).

Examples


```
{
    "bucket_stats": [
      0.0,
      3.0,
      24.0,
      81.0,
      192.0,
      375.0
    ],
    "per_key": "structure-unit-size",
    "uid": "pow::test3",
    "x_end": 5,
    "r_square": 1.0,
    "method": "moving_average",
    "moving_method": "sma",
    "x_start": 0,
    "window_width": 1
}
```


The example above shows an example of profile post-processed by moving average postprocessor (note that this in only an excerpt of the whole profile). Each such model of moving average model shows the computed values, that are represented by *bucket_stats*. The important role has value *moving_method*, that represents the method, which was used to create this model. In this field may be one from the following shortcuts *SMA*, *SMM*, *EMA*, which represents above described methods. The value *r_square* serves to assess the suitability of the model and represents the *coefficient of determination* (R^2). Another significant value in the context of the information about the moving average models is the *window_width*. This value represents the width of the window, that was used at creating this model. Since each model can be used in the further interpretation (either by *Scatter Plot* or *Average Amount Threshold*), another values have auxiliary character and serves for a different purposes at its interpretation. Additional values that contain the information about postprocess parameters can be found in the whole profile, specifically in the part about used post-processors.



The *Scatter Plot* above shows the interpreted model, computed using the **exponential moving average** method, running with default values of parameters. In the picture, one can see that the dependency of running time based on the structural size is best fitted by *exponential* models.

5.1.6 Kernel Regression Methods

A postprocessor that executing the kernel regression over the resources.

Postprocessing of inputs profiles using the kernel regression. Postprocessor, implementing kernel regression offers several computational methods with different approaches and different strategies to find optimal parameters.

Command Line Interface

perun postprocessby kernel-regression

Execution of the interleaving of profiles resources by *kernel* models.

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

In statistics, the kernel regression is a non-parametric approach to estimate the conditional expectation of a random variable. Generally, the main goal of this approach is to find non-parametric relation between a pair of random variables X <per-key> and Y <of-key>. Different from parametric techniques (e.g. linear regression), kernel regression does not assume any underlying distribution (e.g. linear, exponential, etc.) to estimate the regression function. The main idea of kernel regression is putting the **kernel**, that have the role of weighted function, to each observation point in the dataset. Subsequently, the kernel will assign weight to each point in depends on the distance from the current data point. The kernel basis formula depends only to the *bandwidth* from the current ('local') data point X to a set of neighboring data points X.

Kernel Selection does not important from an asymptotic point of view. It is appropriate to choose the **optimal** kernel since this group of the kernels are continuously on the whole definition field and then the estimated regression function inherit smoothness of the kernel. For example, a suitable kernels can be the **epanechnikov** or **normal** kernel. This postprocessor offers the **kernel selection** in the **kernel-smoothing** mode, where are available five different types of kernels. For more information about these kernels or this kernel regression mode you can see *perun postprocessby kernel-regression kernel-smoothing*.

Bandwidth Selection is the most important factor at each approach of kernel regression, since this value significantly affects the smoothness of the resulting estimate. In case, when is choose the inappropriate value, in the most cases can be expected the following two situations. The **small** bandwidth value reproduce estimated data and vice versa, the **large** value leads to over-leaving, so to average of the estimated data. Therefore are used the methods to determine the bandwidth value. One of the most widespread and most commonly used methods is the **cross-validation** method. This method is based on the estimate of the regression function in which will be omitted *i-th* observation. In this postprocessor is this method available in the **estimator-setting** mode. Another methods to determine the bandwidth, which are available in the remaining modes of this postprocessor are **scott** and **silverman** method. More information about these methods and its definition you cas see in the part *perun postprocessby kernel-regression method-selection*.

This postprocessor in summary offers five different modes, which does not differ in the resulting estimate, but in the way of computation the resulting estimate. Better said, it means, that the result of each mode is the **kernel estimate** with relevant parameters, selected according to the concrete mode. In short we will describe the individual methods, for more information about it, you can visit the relevant parts of documentation:

* **Estimator-Settings**: Nadaraya-Watson kernel regression with specific settings for estimate
* **User-Selection**: Nadaraya-Watson kernel regression with user bandwidth
* **Method-Selection**: Nadaraya-Watson kernel regression with supporting bandwidth selection method
* **Kernel-Smoothing**: Kernel regression with different types of kernel and regression methods
* **Kernel-Ridge**: Nadaraya-Watson kernel regression with automatic bandwidth selection

For more details about this approach of non-parametric analysis refer to *Kernel Regression Methods*.

```
perun postprocessby kernel-regression [OPTIONS] COMMAND [ARGS]...
```

Options

-of, --of-key <of_key>
Sets key for which we are finding the model (y-coordinates).

-per, --per-key <per_key>
Sets the key that will be used as a source variable (x-coordinates).

Commands

estimator-settings

kernel-ridge

kernel-smoothing

method-selection

user-selection

perun postprocessby kernel-regression estimator-settings

Nadaraya-Watson kernel regression with specific settings for estimate.

As has been mentioned above, the kernel regression aims to estimate the functional relation between explanatory variable **y** and the response variable **X**. This mode of kernel regression postprocessor calculates the conditional mean **E[y|X] = m(X)**, where **y = m(X) +** ϵ. Variable **X** is represented in the postprocessor by <per-key> option and the variable **y** is represented by <of-key> option.

Regression Estimator <reg-type>:

> This mode offer two types of *regression estimator <reg-type>*. *Local Constant ('ll')* type of regression provided by this mode is also known as *Nadaraya-Watson* kernel regression:
>
> > **Nadaraya-Watson**: expects the following conditional expectation: **E[y|X] = m(X)**, where function **m(*)** represents the regression function to estimate. Then we can alternatively write the following formula: **y = m(X) +** ϵ, **E** (ϵ) **= 0**. Then we can suppose, that we have the set of independent observations $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ and the **Nadaraya-Watson** estimator is defined as:
>
> $$m_h(x) = \sum_{i=1}^{n} K_h(x - x_i) y_i / \sum_{j=1}^{n} K_h(x - x_j)$$
>
> > where K_h is a kernel with bandwidth *h*. The denominator is a weighting term with sum 1. It easy to see that this kernel regression estimator is just a weighted sum of the observed responses y_i. There are many other kernel estimators that are various in compare to this presented estimator. However, since all are asymptotic equivalently, we will not deal with them closer. **Kernel Regression** postprocessor works in all modes only with **Nadaraya-Watson** estimator.
>
> The second supported *regression estimator* in this mode of postprocessor is *Local Linear ('lc')*. This type is an extension of that which suffers less from bias issues at the edge of the support.
>
> > **Local Linear**: estimator, that offers various advantages compared with other kernel-type estimators, such as the *Nadaraya-Watson* estimator. More precisely, it adapts to both random and fixed designs, and to various design densities such as highly clustered designs and nearly uniform designs. It turns out that the *local linear* smoother repairs the drawbacks of other kernel regression estimators. An regression estimator *m* of *m* is a linear smoother if, for each *x*, there is a vector $l(x) = (l_1(x), ..., l_n(x))^T$ such that:
>
> $$m(x) = \sum_{i=1}^{n} l_i(x) Y_i = l(x)^T Y$$
>
> > where $Y = (Y_1, ..., Y_n)^T$. For kernel estimators:
>
> $$l_i(x) = K(||x - X_i||/h) / \sum_{j=1}^{n} K(||x - X_j||/h)$$
>
> > where *K* represents kernel and *h* its bandwidth.
>
> For a better imagination, there is an interesting fact, that the following estimators are linear smoothers too: *Gaussian process regression*, *splines*.

Bandwidth Method <bandwidth-method>:

As has been said in the general description of the *kernel regression*, once of the most important factors of the resulting estimate is the kernel **bandwidth**. When the inappropriate value is selected may occur to *under-laying* or *over-laying* fo the resulting kernel estimate. Since the bandwidth of the kernel is a free parameter which exhibits a strong influence on the resulting estimate postprocessor offers the method for its selection. Two most popular data-driven methods of bandwidth selection that have desirable properties are *least-squares cross-validation* (*cv_ls*) and the *AIC-based* method of *Hurvich et al. (1998)*, which is based on minimizing a modified *Akaike Information Criterion* (*aic*):

Cross-Validation Least-Squares: determination of the optimal kernel bandwidth for kernel regression is based on minimizing

$$CV(h) = n^{-1} \sum_{i=1}^{n} (Y_i - g_{-i}(X_i))^2,$$

where $g_{-i}(X_i)$ is the estimator of $g(X_i)$ formed by leaving out the *i-th* observation when generating the prediction for observation *i*.

Hurvich et al.'s (1998) approach is based on the minimization of

$$AIC_c = ln(\sigma^2) + ((1 + tr(H)/n)/(1 - (tr(H) + 2)/n),$$

where

$$\sigma^2 = 1/n \sum_{i=1}^{n} (Y_i - g(X_i))^2 = Y'(I - H)'(I - H)Y/n$$

with $g(X_i)$ being a non-parametric regression estimator and *H* being an *n x n* matrix of kernel weights with its *(i, j)-th* element given by $H_{ij} = K_h(X_i, X_j)/\sum_{l=1}^{n} K_h(X_i, X_l)$, where $K_h(*)$ is a generalized product kernel.

Both methods for kernel bandwidth selection the *least-squared cross-validation* and the *AIC* have been shown to be asymptotically equivalent.

The remaining options at this mode of kernel regression postprocessor are described within usage to it and you can see this in the list below. All these options are parameters to *EstimatorSettings* (see EstimatorSettings), that optimizing the kernel bandwidth based on the these specified settings.

In the case of confusion about this approach of kernel regression, you can visit StatsModels.

```
perun postprocessby kernel-regression estimator-settings [OPTIONS]
```

Options

-rt, --reg-type <reg_type>
: Provides the type for regression estimator. Supported types are: "lc": local-constant (Nadaraya-Watson) and "ll": local-linear estimator. Default is "ll". For more information about these types you can visit Perun Documentation.

-bw, --bandwidth-method <bandwidth_method>
: Provides the method for bandwidth selection. Supported values are: "cv-ls": least-squares cross validation and "aic": AIC Hurvich bandwidth estimation. Default is "cv-ls". For more information about these methods you can visit Perun Documentation.

--efficient, --uniformly
: If True, is executing the efficient bandwidth estimation - by taking smaller sub-samples and estimating the scaling factor of each sub-sample. It is useful for large samples and/or multiple variables. If False (default), all data is used at the same time.

--randomize, --no-randomize
If True, the bandwidth estimation is performed by taking <n_res> random re-samples of size <n-sub-samples> from the full sample. If set to False (default), is performed by slicing the full sample in sub-samples of <n-sub-samples> size, so that all samples are used once.

-nsub, --n-sub-samples <n_sub_samples>
Size of the sub-samples (default is 50).

-nres, --n-re-samples <n_re_samples>
The number of random re-samples used to bandwidth estimation. It has effect only if <randomize> is set to True. Default values is 25.

--return-median, --return-mean
If True, the estimator uses the median of all scaling factors for each sub-sample to estimate bandwidth of the full sample. If False (default), the estimator used the mean.

perun postprocessby kernel-regression user-selection

Nadaraya-Watson kernel regression with user bandwidth.

This mode of kernel regression postprocessor is very similar to *estimator-settings* mode. Also offers two types of *regression estimator* <reg-type> and that the *Nadaraya-Watson* estimator, so known as *local-constant* (*lc*) and the *local-linear* estimator (*ll*). Details about these estimators are available in *perun postprocessby kernel-regression estimator-settings*. In contrary to this mode, which selected a kernel bandwidth using the *EstimatorSettings* and chosen parameters, in this mode the user itself selects a kernel bandwidth <bandwidth-value>. This value will be used to execute the kernel regression. The value of kernel bandwidth in the resulting estimate may change occasionally, specifically in the case, when the bandwidth value is too low to execute the kernel regression. Then will be a bandwidth value approximated to the closest appropriate value, so that is not decreased the accuracy of the resulting estimate.

```
perun postprocessby kernel-regression user-selection [OPTIONS]
```

Options

-rt, --reg-type <reg_type>
Provides the type for regression estimator. Supported types are: “lc”: local-constant (Nadaraya-Watson) and “ll”: local-linear estimator. Default is “ll”. For more information about these types you can visit Perun Documentation.

-bv, --bandwidth-value <bandwidth_value>
The float value of <bandwidth> defined by user, which will be used at kernel regression. [required]

perun postprocessby kernel-regression method-selection

Nadaraya-Watson kernel regression with supporting bandwidth selection method.

The last method from a group of three methods based on a similar principle. *Method-selection* mode offers the same type of *regression estimators* <reg-type> as the first two described methods. The first supported option is *ll*, which represents the *local-linear* estimator. *Nadaraya-Watson* or *local constant* estimator represents the second option for <reg-type> parameter. The more detailed description of these estimators is located in *perun postprocessby kernel-regression estimator-settings*. The difference between this mode and the two first modes is in the way of determination of a kernel bandwidth. In this mode are offered two methods to determine bandwidth. These methods try calculated an optimal bandwidth from predefined formulas:

Scotts's Rule of thumb to determine the smoothing bandwidth for a kernel estimation. It is very fast compute. This rule was designed for density estimation but is usable for kernel regression too. Typically produces a larger bandwidth and therefore it is useful for estimating a gradual trend:

$$bw = 1.059 * A * n^{-1/5},$$

where *n* marks the length of X variable <per-key> and

$$A = min(\sigma(x), IQR(x)/1.349),$$

where σ marks the StandardDeviation and IQR marks the InterquartileRange.

Silverman's Rule of thumb to determine the smoothing bandwidth for a kernel estimation. Belongs to most popular method which uses the *rule-of-thumb*. Rule is originally designs for *density estimation* and therefore uses the normal density as a prior for approximating. For the necessary estimation of the σ of X <per-key> he proposes a robust version making use of the InterquartileRange. If the true density is uni-modal, fairly symmetric and does not have fat tails, it works fine:

$$bw = 0.9 * A * n^{-1/5},$$

where *n* marks the length of X variable <per-key> and

$$A = min(\sigma(x), IQR(x)/1.349),$$

where σ marks the StandardDeviation and IQR marks the InterquartileRange.

```
perun postprocessby kernel-regression method-selection [OPTIONS]
```

Options

-rt, --reg-type <reg_type>
Provides the type for regression estimator. Supported types are: "lc": local-constant (Nadaraya-Watson) and "ll": local-linear estimator. Default is "ll". For more information about these types you can visit Perun Documentation.

-bm, --bandwidth-method <bandwidth_method>
Provides the helper method to determine the kernel bandwidth. The <method_name> will be used to compute the bandwidth, which will be used at kernel regression.

perun postprocessby kernel-regression kernel-smoothing

Kernel regression with different types of kernel and regression methods.

This mode of kernel regression postprocessor implements non-parametric regression using different kernel methods and different kernel types. The calculation in this mode can be split into three parts. The first part is represented by the *kernel type*, the second part by *bandwidth computation* and the last part is represented by *regression method*, which will be used to interleave the given resources. We will look gradually at individual supported options in the each part of computation.

Kernel Type <kernel-type>:

In non-parametric statistics a *kernel* is a weighting function used in estimation techniques. In *kernel regression* is used to estimate the conditional expectation of a random variable. As has been said, *kernel*

width must be specified when running a non-parametric estimation. The *kernel* in view of mathematical definition is a non-negative real-valued integrable function *K*. For most applications, it is desirable to define the function to satisfy two additional requirements:

Normalization:

$$\int_{-\infty}^{+\infty} K(u)du = 1,$$

Symmetry

$$K(-u) = K(u),$$

for all values of u. The second requirement ensures that the average of the corresponding distribution is equal to that of the sample used. If *K* is a kernel, then so is the function K^* defined by $K^*(u) = \lambda K(\lambda u)$, where $\lambda > 0$. This can be used to select a scale that is appropriate for the data. This mode offers several types of kernel functions:

Kernel Name	Kernel Function, K(u)	Efficiency
Gaussian (normal)	$K(u) = (1/\sqrt{2\pi})e^{-(1/2)u^2}$	95.1%
Epanechnikov	$K(u) = 3/4(1 - u^2)$	100%
Tricube	$K(u) = 70/81(1 - \lvert u^3\rvert)^3$	99.8%
Gaussian order4	$\phi_4(u) = 1/2(3 - u^2)\phi(u)$, where ϕ is the normal kernel	not applicable
Epanechnikov order4	$K_4(u) = -(15/8)u^2 + (9/8)$, where *K* is the non-normalized Epanechnikov kernel	not applicable

Efficiency is defined as $\sqrt{\int u^2 K(u)du \int K(u)^2 du}$, and its measured to the *Epanechnikov* kernel.

Smoothing Method <smoothing-method>:

Kernel-Smoothing mode of this postprocessor offers three different non-parametric regression methods to execute *kernel regression*. The first of them is called *spatial-average* and perform a *Nadaraya-Watson* regression (i.e. also called local-constant regression) on the data using a given kernel:

$$m_h(x) = \sum_{i=1}^{n} K_h((x - x_i)/h)y_i / \sum_{j=1}^{n} K_h((x - x_j)/h),$$

where *K(x)* is the kernel and must be such that *E(K(x)) = 0* and *h* is the bandwidth of the method. *Local-Constant* regression was also described in *perun postprocessby kernel-regression estimator-settings*. The second supported regression method by this mode is called *local-linear*. Compared with previous method, which offers computational with different types of kernel, this method has restrictions and perform *local-linear* regression using only *Gaussian (Normal)* kernel. The *local-constant* regression was described in *perun postprocessby kernel-regression estimator-settings* and therefore will not be given no further attention to it. *Local Polynomial* regression is the last method in this mode and perform regression in *N-D* using a user-provided kernel. The *local-polynomial* regression is the function that minimizes, for each position:

$$m_h(x) = \sum_{i=0}^{n} K((x - x_i)/h)(y_i - a_0 - P_q(x_i - x))^2,$$

where *K(x)* is the kernel such that *E(K(x)) = 0*, *q* is the order of the fitted polynomial <polynomial-order>, $P_q(x)$ is a polynomial or order *q* in *x*, and *h* is the bandwidth of the method. The polynomial $P_q(x)$ is of the form:

$$F_d(k) = n \in N^d | \sum_{i=1}^{d} n_i = k$$

$$P_q(x_1, ..., x_d) = \sum_{k=1}^{q} \sum_{n \in F_d(k)} a_{k,n} \prod_{i=1}^{d} x_i^{n_i}$$

For example we can have:

$$P_2(x, y) = a_{110}x + a_{101}y + a_{220}x^2 + a_{221}xy + a_{202}y^2$$

The last part of the calculation is the *bandwidth* computation. This mode offers to user enter the value directly with use of parameter <bandwidth-value>. The parameter <bandwidth-method> offers to user the selection from the two methods to determine the optimal bandwidth value. The supported methods are *Scotts's Rule* and *Silverman's Rule*, which are described in *perun postprocessby kernel-regression method-selection*. This parameter cannot be entered in combination with <bandwidth-value>, then will be ignored and will be accepted value from <bandwidth-value>.

```
perun postprocessby kernel-regression kernel-smoothing [OPTIONS]
```

Options

-kt, --kernel-type <kernel_type>
Provides the set of kernels to execute the *kernel-smoothing* with kernel selected by the user. For exact definitions of these kernels and more information about it, you can visit the Perun Documentation.

-sm, --smoothing-method <smoothing_method>
Provides kernel smoothing methods to executing non-parametric regressions: *local-polynomial* perform a local-polynomial regression in N-D using a user-provided kernel; *local-linear* perform a local-linear regression using a gaussian (normal) kernel; and *spatial-average* perform a Nadaraya-Watson regression on the data (so called local-constant regression) using a user-provided kernel.

-bm, --bandwidth-method <bandwidth_method>
Provides the helper method to determine the kernel bandwidth. The <bandwidth_method> will be used to compute the bandwidth, which will be used at kernel-smoothing regression. Cannot be entered in combination with <bandwidth-value>, then will be ignored and will be accepted value from <bandwidth-value>.

-bv, --bandwidth-value <bandwidth_value>
The float value of <bandwidth> defined by user, which will be used at kernel regression. If is entered in the combination with <bandwidth-method>, then method will be ignored.

-q, --polynomial-order <polynomial_order>
Provides order of the polynomial to fit. Default value of the order is equal to 3. Is accepted only by *local-polynomial* <smoothing-method>, another methods ignoring it.

perun postprocessby kernel-regression kernel-ridge

Nadaraya-Watson kernel regression with automatic bandwidth selection.

This mode implements *Nadaraya-Watson* kernel regression, which was described above in *perun postprocessby kernel-regression estimator-settings*. While the previous modes provided the methods to determine the optimal bandwidth with different ways, this method provides a little bit different way. From a given range of potential bandwidths <gamma-range> try to select the optimal kernel bandwidth with use of *leave-one-out cross-validation*. This approach was described in *perun postprocessby kernel-regression estimator-settings*, where was introduced the *least-squares cross- validation* and it is a modification of this approach. *Leave-one-out cross validation* is K-fold cross validation taken to its logical extreme, with *K* equal to *N*, the number of data points in the set. The original *gamma-range* will be divided on the base of size the given step <gamma-step>. The selection of specific value from this range will be executing by minimizing mean-squared-error in *leave-one-out cross-validation*. The selected *bandwidth-value* will serves for *gaussian* kernel in resulting estimate: $K(x, y) = exp(-gamma * ||x - y||^2)$.

```
perun postprocessby kernel-regression kernel-ridge [OPTIONS]
```

Options

-gr, --gamma-range <gamma_range>
: Provides the range for automatic bandwidth selection of the kernel via leave-one-out cross-validation. One value from these range will be selected with minimizing the mean-squared error of leave-one-out cross-validation. The first value will be taken as the lower bound of the range and cannot be greater than the second value.

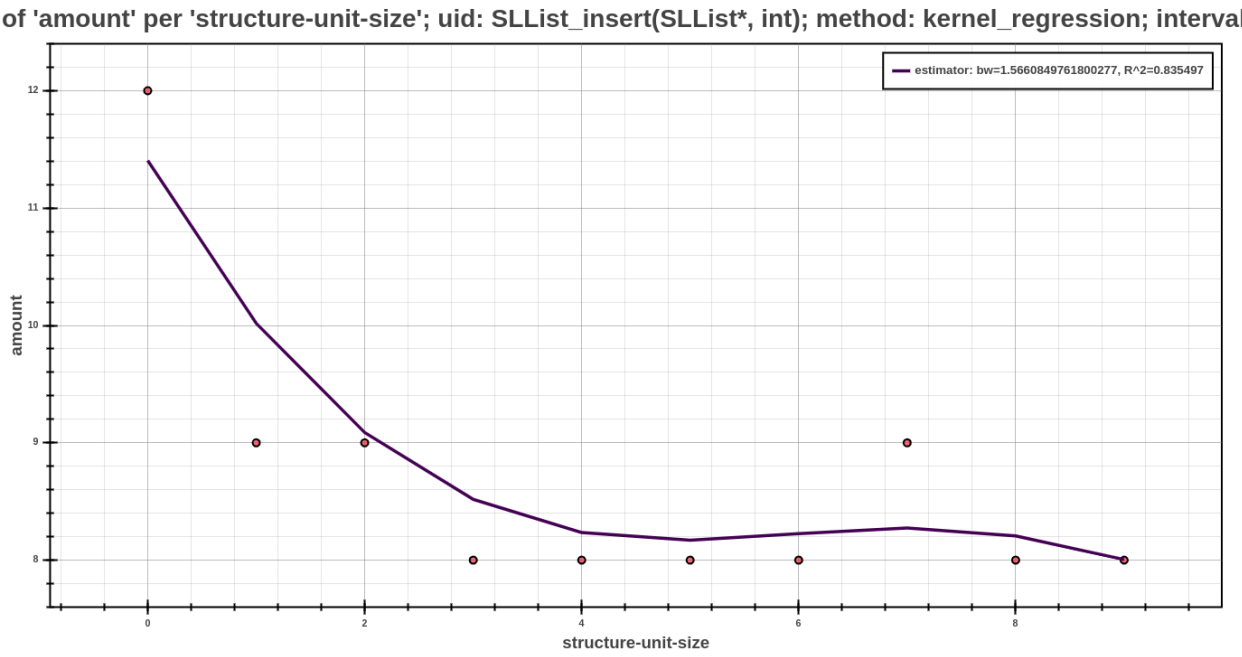
-gs, --gamma-step <gamma_step>
: Provides the size of the step, with which will be executed the iteration over the given <gamma-range>. Cannot be greater than length of <gamma-range>, else will be set to value of the lower bound of the <gamma_range>.

Examples


```
{
    "per_key": "structure-unit-size",
    "uid": "quad::test1",
    "kernel_mode": "estimator",
    "r_square": 0.9990518378010778,
    "method": "kernel_regression",
    "x_start": 10,
    "bandwidth": 2.672754640321602,
    "x_end": 64,
    "kernel_stats": [
          115.6085941489687,
          155.95838478107163,
          190.27598428091824,
          219.36576520977312,
          252.80699243117965,
          268.4600214673941,
          283.3744716372719,
          282.7535719770607,
          276.27153279181573,
          269.69580474542016,
          244.451017529157,
          226.98819185034756,
          180.72465187812492
```

```
    ]
}
```

The example above shows an example of profile post-processed by *kernel regression* (note that this is only an excerpt of the whole profile). Each such kernel model shows the values of resulting kernel estimate, that are part of *kernel_stats* list. Another fascinating value is stored in *kernel_mode* field and means the relevant mode, which executing the *kernel regression* over this model. In this field may be one from the following words, which represents the individual modes of kernel regression postprocessor. The value *r_square* serves to assess the suitability of the kernel model and represents the *coefficient of determination* (R^2). In the context of another kernel estimates for decreasing or increasing the resulting accuracy is important the field *bandwidth*, which represents the kernel bandwidth in the current kernel model. Since each model can be used in the further interpretation (either by *Scatter Plot* or *Average Amount Threshold*), another values have auxiliary character and serves for a different purposes at its interpretation. Additional values that contain the information about selected parameters at kernel regression postprocessor and its modes, can be found in the whole profile, specifically in the part about used post-processors.



The *Scatter Plot* above shows the interpreted model, computed using the *kernel regression* postprocessor, concretely with default value of parameters in **estimator-settings** mode of this postprocessor. In the picture, can be see that the dependency of running time based on the structural size.

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 post-processing 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/e/phdwork/perun/git/docs/_static/templates/postprocess_run.py
+++ /mnt/e/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/e/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/e/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪postprocess.py
@@ -7,6 +7,6 @@
     return {
         'vcs': ['git'],
         'collect': ['trace', '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.

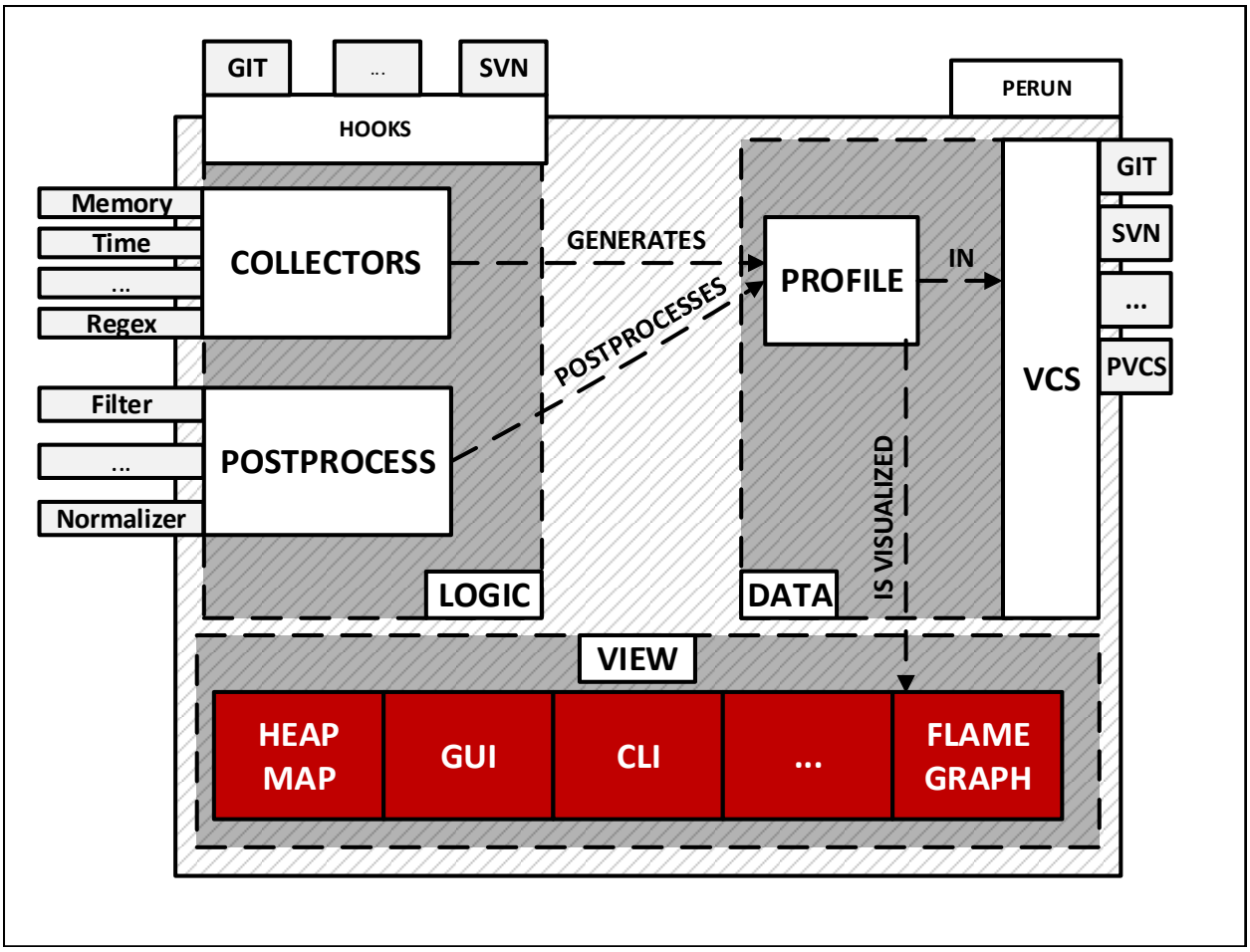
The format of input profiles has to be w.r.t. *Specification of Profile Format*, in particular the intepreted profiles should contain the `resources` region with data.

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.

5. *Table Of* transforms either the resources or models of the profile into a tabular representation. The table can be further modified by (1) changing the format (see tabulate for table formats), (2) limiting rows or columns displayed, or (3) sorting w.r.t specified keys.

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

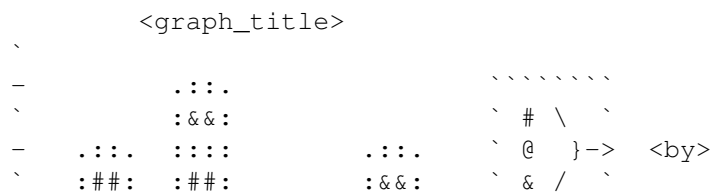
Bars graph shows the aggregation (e.g. sum, count, etc.) of resources of given types (or keys). Each bar shows `<func>` of resources from `<of>` key (e.g. sum of amounts, average of amounts, count of types, etc.) per each `<per>` key (e.g. per each snapshot, or per each type). Moreover, the graphs can 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>`.

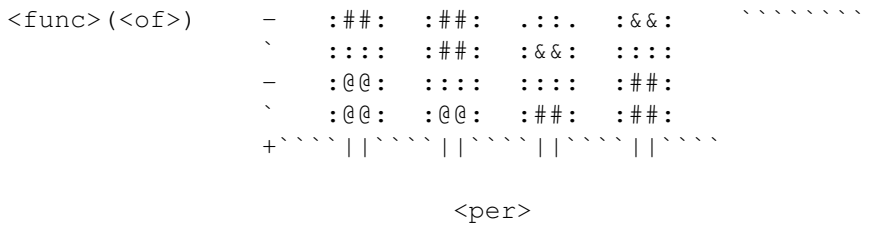
Bokeh library is the current interpretation backend, which generates HTML files, that can be opened directly in the browser. Resulting graphs can be further customized by adding custom labels for axes, custom graph title or different graph width.

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:





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.

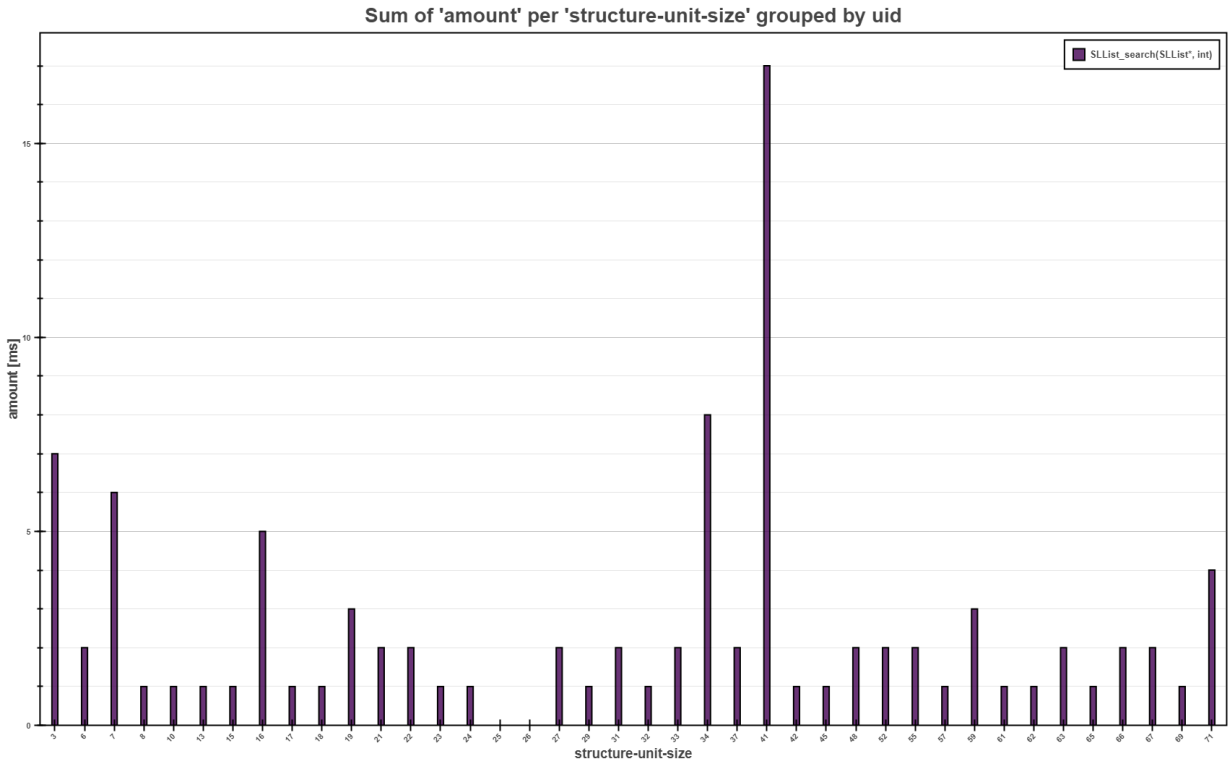
-gt, --graph-title <graph_title>
Sets the custom title of the bars graph.

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

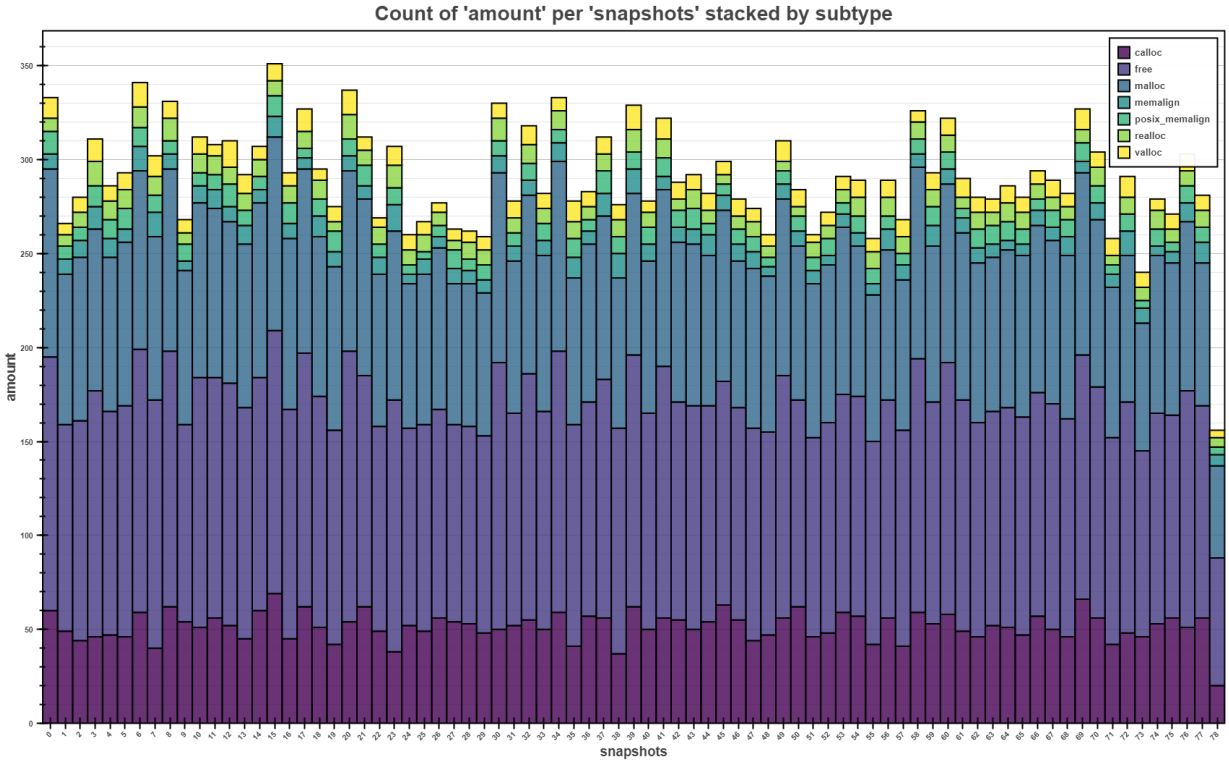
Arguments

<aggregation_function>
Optional argument

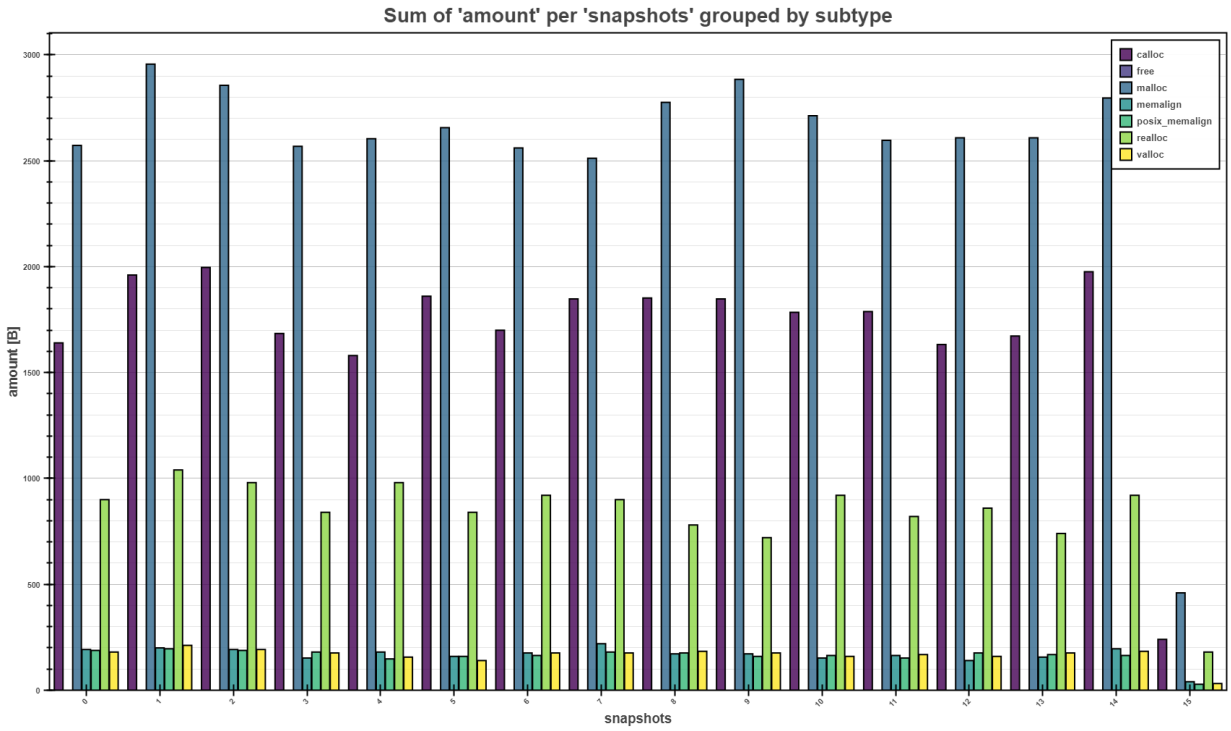
Examples of Output



The *Bars Plot* above shows the overall sum of the running times for each `structure-unit-size` for the `SLList_search` function collected by *Trace 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

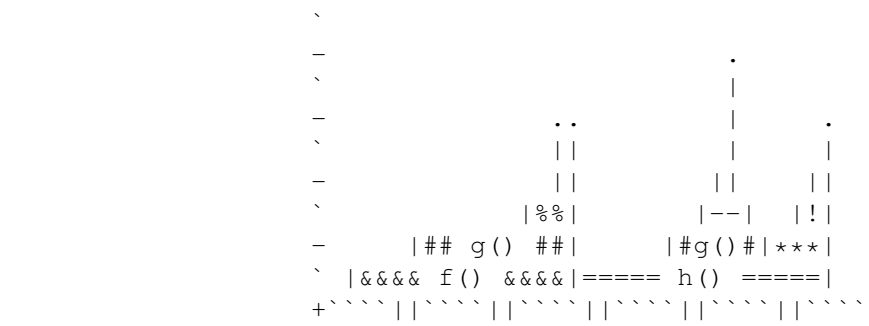
Flame graph interprets the relative and inclusive presence of the resources according to the stack depth of the origin of resources.

- **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:



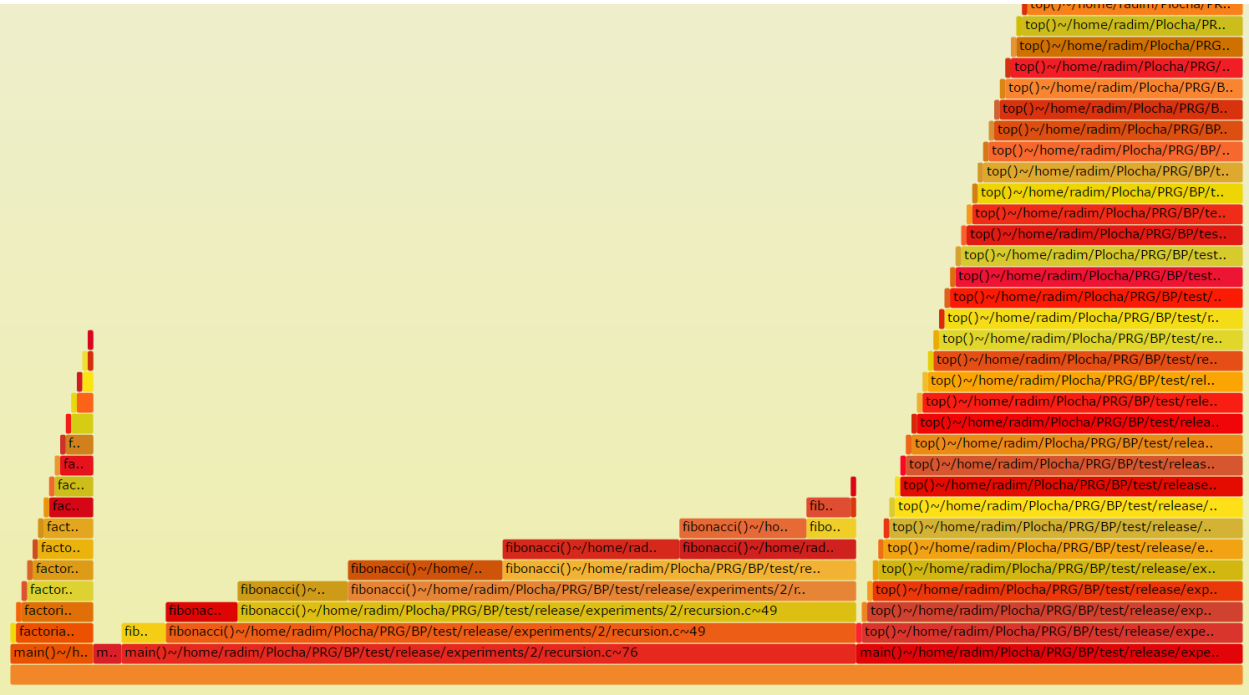
Refer to *Flame Graph* for more thorough description and examples of the interpretation technique. Refer to `perun.profile.convert.to_flame_graph_format()` for more details how the profiles are converted to the flame graph format.

```
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

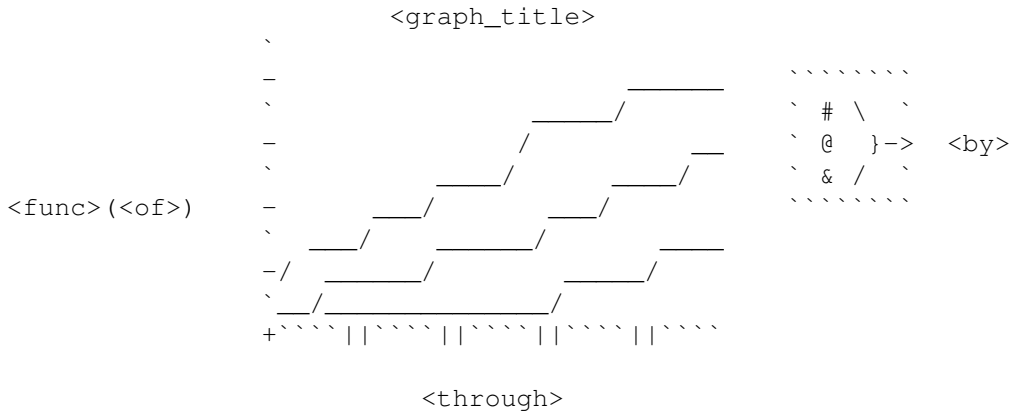
Flow graph shows the values resources depending on the independent variable as basic graph. For each group of resources identified by unique value of `<by>` key, one graph shows the dependency of `<of>` values aggregated by `<func>` depending on the `<through>` key. Moreover, the values can either be 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.

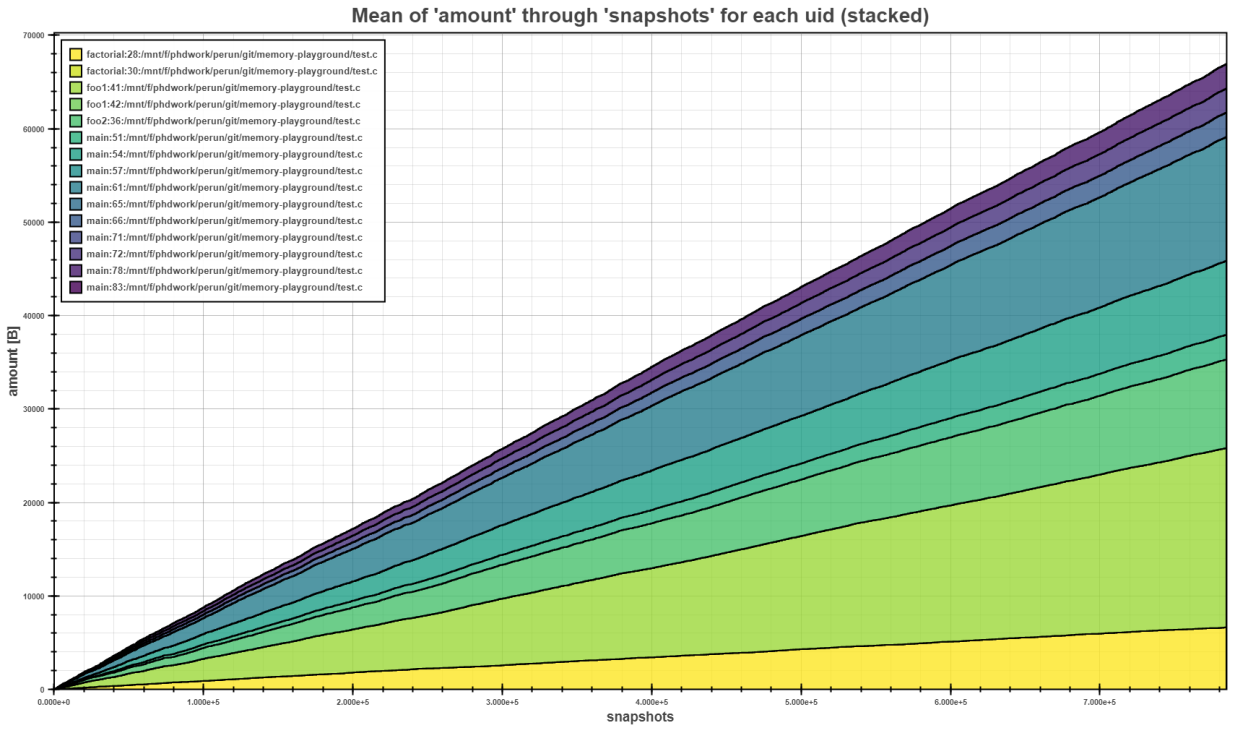
-gt, --graph-title <graph_title>
: Sets the custom title of the flow graph.

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

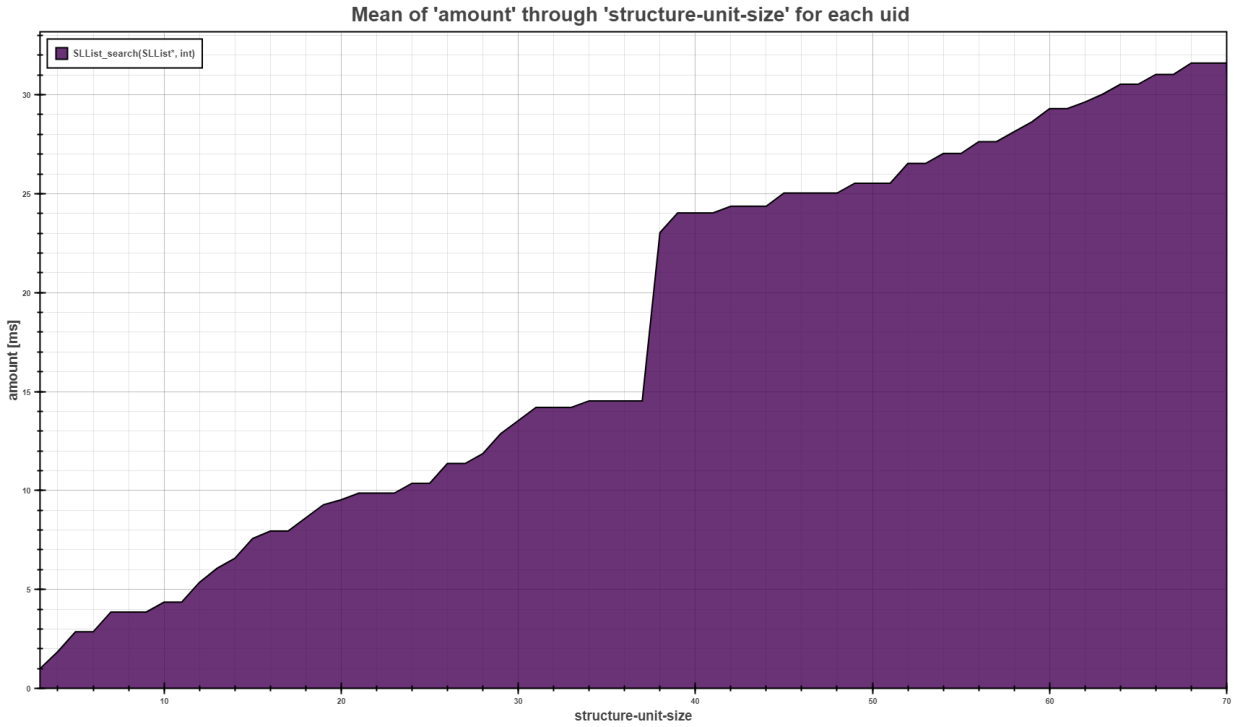
Arguments

<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.

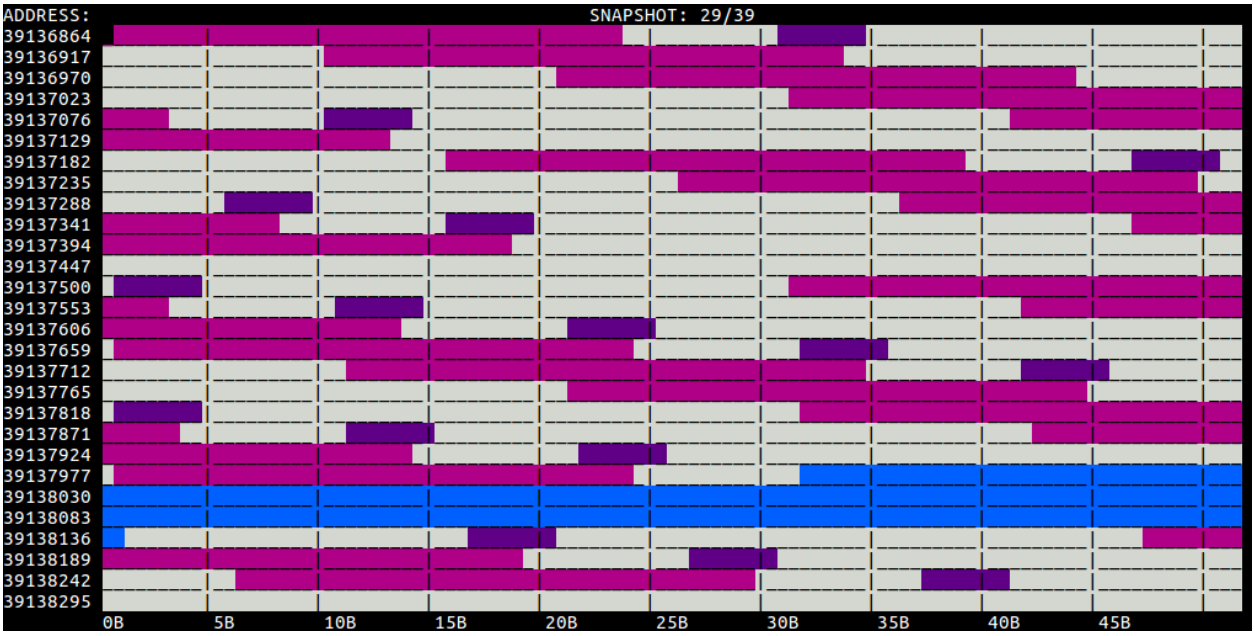
Moreover, the heap map contains *heat map* mode, which accumulates the allocations into the heat representation—the hotter the colour displayed at given memory cell, the more time it was allocated there.

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.

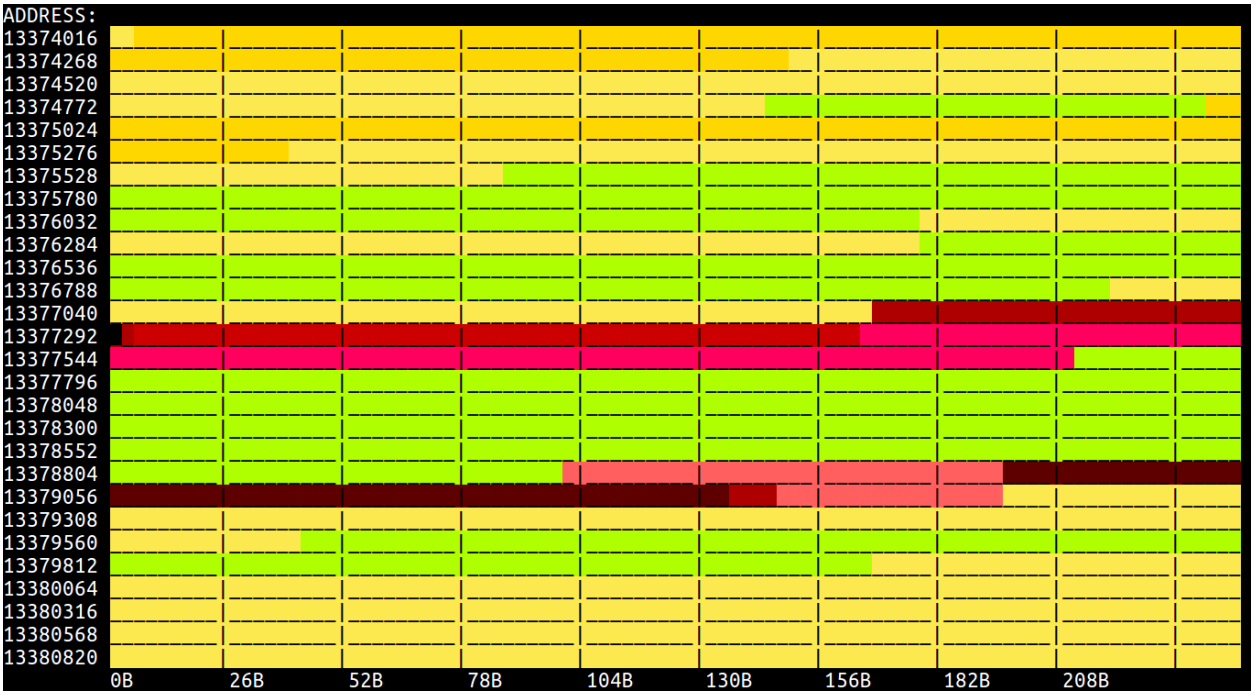
Refer to *Heap Map* for more thorough description and example of *heapmap* interpretation possibilities.

```
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.

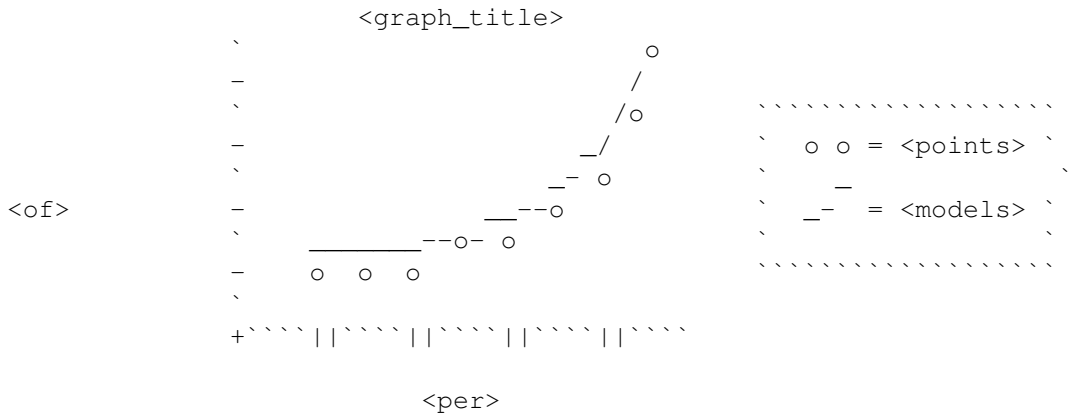
- **Limitations**: *none*.
- **Interpretation style**: graphical
- **Visualization backend**: Bokeh

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.

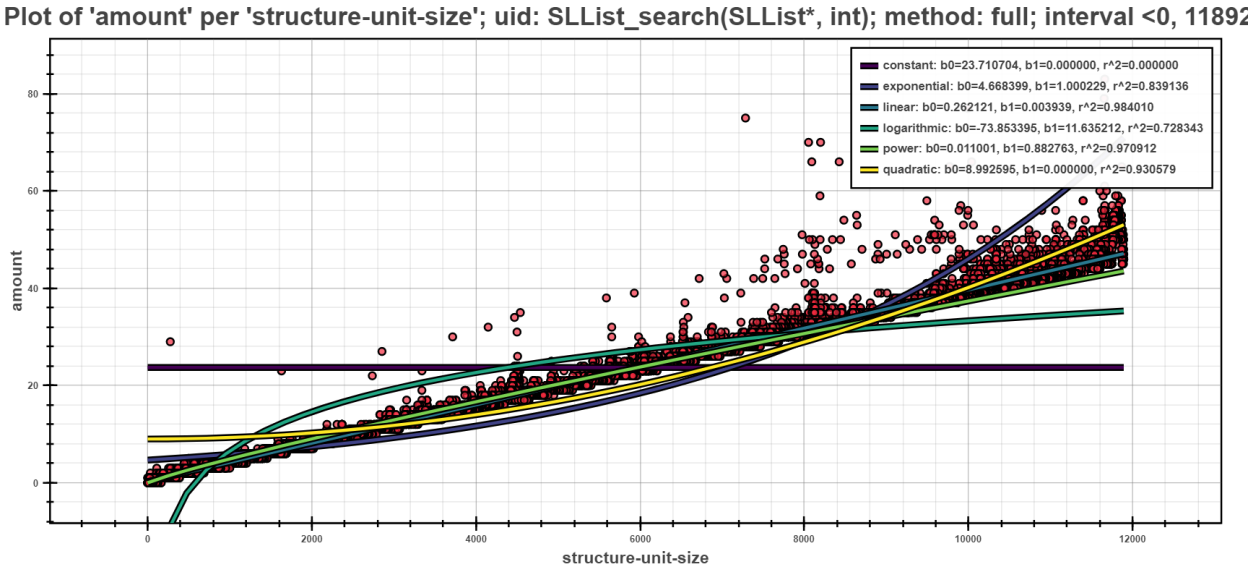
-xl, --x-axis-label <x_axis_label>
Label on the X axis of the scatter plot.

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

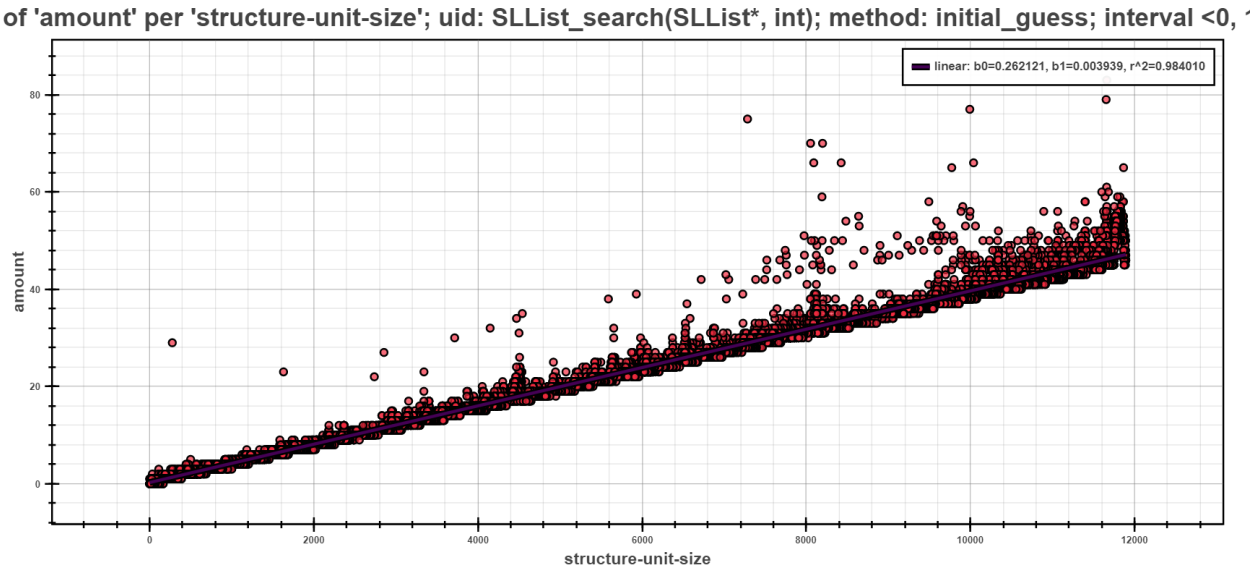
-gt, --graph-title <graph_title>
Title of the scatter plot.

-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.

6.1.6 Table Of

Table interprets the data as a two dimensional array. The cells in the table can be limited to certain columns only and allow output to the file.

Overview and Command Line Interface

perun show tableof

Textual representation of the profile as a table.

- **Limitations**: *none*.
- **Interpretation style**: textual
- **Visualization backend**: tabulate

The table is formatted using the tabulate library. Currently, we support only the simplest form, and allow output to file.

The example output of the tableof is as follows:

```
uid                          model          r_square
---------------------------  -----------  -----------
SLList_insert(SLList*, int)  logarithmic  0.000870412
SLList_insert(SLList*, int)  linear       0.001756
```

```
    SLList_insert(SLList*, int)  quadratic    0.00199925
    SLList_insert(SLList*, int)  power        0.00348063
    SLList_insert(SLList*, int)  exponential  0.00707644
    SLList_search(SLList*, int)  constant     0.0114714
    SLList_search(SLList*, int)  logarithmic  0.728343
    SLList_search(SLList*, int)  exponential  0.839136
    SLList_search(SLList*, int)  power        0.970912
    SLList_search(SLList*, int)  linear       0.98401
    SLList_search(SLList*, int)  quadratic    0.984263
    SLList_insert(SLList*, int)  constant     1
```

Refer to views-table for more thorough description and example of *table* interpretation possibilities.

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

Options

-tf, --to-file
The table will be saved into a file. By default, the name of the output file is automatically generated, unless *–output-file* option does not specify the name of the output file.

-ts, --to-stdout
The table will be output to standard output.

-of, --output-file <output_file>
Target output file, where the transformed table will be saved.

-f, --format <tablefmt>
Format of the outputted table

Commands

models

resources

perun show tableof resources

Outputs the resources of the profile as a table

```
perun show tableof resources [OPTIONS]
```

Options

-h, --headers <headers>
Sets the headers that will be displayed in the table. If none are stated then all of the headers will be outputed

-s, --sort-by <sort_by>
Sorts the table by <key>.

-f, --filter-by <filter_by>
Filters the table to rows, where <key> == <value>. If the *–filter* is set several times, then rows satisfying all rules will be selected for different keys; and the rows satisfying some rule will be sellected for same key.

perun show tableof models

Outputs the models of the profile as a table

```
perun show tableof models [OPTIONS]
```

Options

-h, --headers <headers>
Sets the headers that will be displayed in the table. If none are stated then all of the headers will be outputed

-s, --sort-by <sort_by>
Sorts the table by <key>.

-f, --filter-by <filter_by>
Filters the table to rows, where <key> == <value>. If the *–filter* is set several times, then rows satisfying all rules will be selected for different keys; and the rows satisfying some rule will be sellected for same key.

Examples of Output

In the following, we show several outputs of the *Table Of*.

```
uid            model           coeffs:b1    coeffs:b0     r_square
-------------  -----------  ------------  -----------  -----------
SLList_insert  logarithmic   0.0240681      0.362624   0.000870412
SLList_insert  linear        9.93516e-06    0.505375   0.001756
SLList_insert  power         0.00978141     0.93533    0.00348063
SLList_insert  exponential   1              0.990979   0.00707644
SLList_search  constant      0             23.7107     0.0114714
SLList_search  logarithmic  11.6352       -73.8534     0.728343
SLList_search  exponential   1.00023        4.6684     0.839136
SLList_search  power         0.882763       0.0110015  0.970912
SLList_search  linear        0.00393897     0.262121   0.98401
SLList_insert  constant      0              0.56445    1
```

The table above shows list of models for *SLList_insert* and *SLList_search* functions for Singly-linked List implementation of test complexity repository sorted by the value of coefficient of determination *r_square* (see *Regression Analysis* for more details about models and coefficient of determination). For each type of model (e.g. linear) we list the value of its coefficients (e.g. for linear function *b1* corresponds to the slope of the function and *b0* to interception of the function).

From the measured data the insert is estimated to be constant and search to be linear.

```
uid:function          class
--------------------  -------
dwtint_decode_strip   O(n^2)
dwtint_decode_band    O(n^2)
dwtint_decode_block   O(n^2)
dwtint_encode_band    O(n^2)
dwtint_encode_block   O(n^2)
dwtint_weight_band    O(n^2)
dwtint_encode_strip   O(n^2)
dwtint_unweight_band  O(n^2)
```

The second table shows list of function with quadratic complexities inferred by collector-bounds for *dwtint.c* module of the CCSDS codec (will be publically available in near future). The rest of the function either could not be inferred

(e.g. due to unsupported construction, or requiring more elaborate static resource bounds analysis—e.g. due to the missing heap analysis) or were linear or constant.

```
\begin{tabular}{llrr}
\hline
 uid              & model   &    coeffs:b1 &    r\_square \\
\hline
 test\_for\_static  & linear  &  3.36163e-08 & 4.46927e-07 \\
 expand\_tag\_fname & linear  &  7.3574e-08  & 2.32882e-06 \\
 vim\_strncpy      & linear  & -1.23499e-08 & 9.71476e-06 \\
 vim\_strsave      & linear  &  2.10692e-08 & 1.10401e-05 \\
 vim\_free         & linear  & -1.15839e-08 & 0.000216828 \\
 parse\_tag\_line   & linear  &  1.04969e-06 & 0.004604    \\
 vim\_strchr       & linear  & -8.84782e-08 & 0.00629398  \\
 skiptowhite      & linear  & -0.00058116  & 0.00714483  \\
 lalloc           & linear  & -1.19402e-07 & 0.0101609   \\
 alloc            & linear  & -1.83755e-07 & 0.0136817   \\
 ga\_grow          & linear  &  3.86559e-06 & 0.0160297   \\
 vim\_regexec      & linear  &  2.398e-06   & 0.0161956   \\
 ga\_clear         & linear  &  0.535088    & 0.0187932   \\
 vim\_strnsave     & linear  & -0.000529842 & 0.0210357   \\
 ga\_init2         & linear  &  0.00913043  & 0.0365217   \\
 test\_for\_current & linear  &  1.00515e-05 & 0.126627    \\
 skipwhite        & linear  &  4.98823e-06 & 0.192099    \\
 vim\_isblankline  & linear  &  8.68793e-06 & 0.195017    \\
 vim\_regfree      & linear  &  5           & 0.75        \\
\hline
\end{tabular}
```

The last example, shows list of estimated linear functions for *vim v7.4.2293* sorted by coefficient of determination *r_square*. The output uses different format (*latex*).

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/e/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/e/phdwork/perun/git/docs/_static/templates/supported_module_names_
↪views.py
@@ -8,5 +8,6 @@
         'vcs': ['git'],
         'collect': ['trace', '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

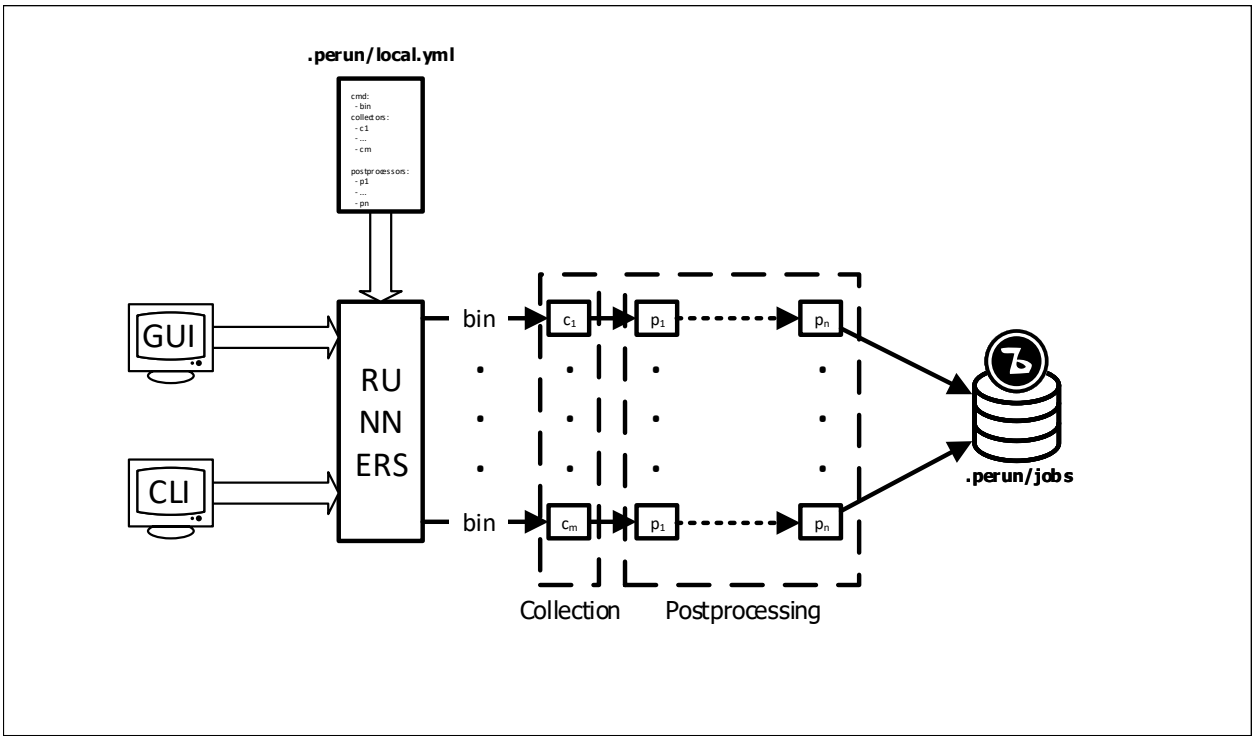
AUTOMATING RUNS

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. If you want to automatically register the newly generated profile into the corresponding minor version index, then set *`profiles.register_after_run`* key to a true value.



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.

7.2 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

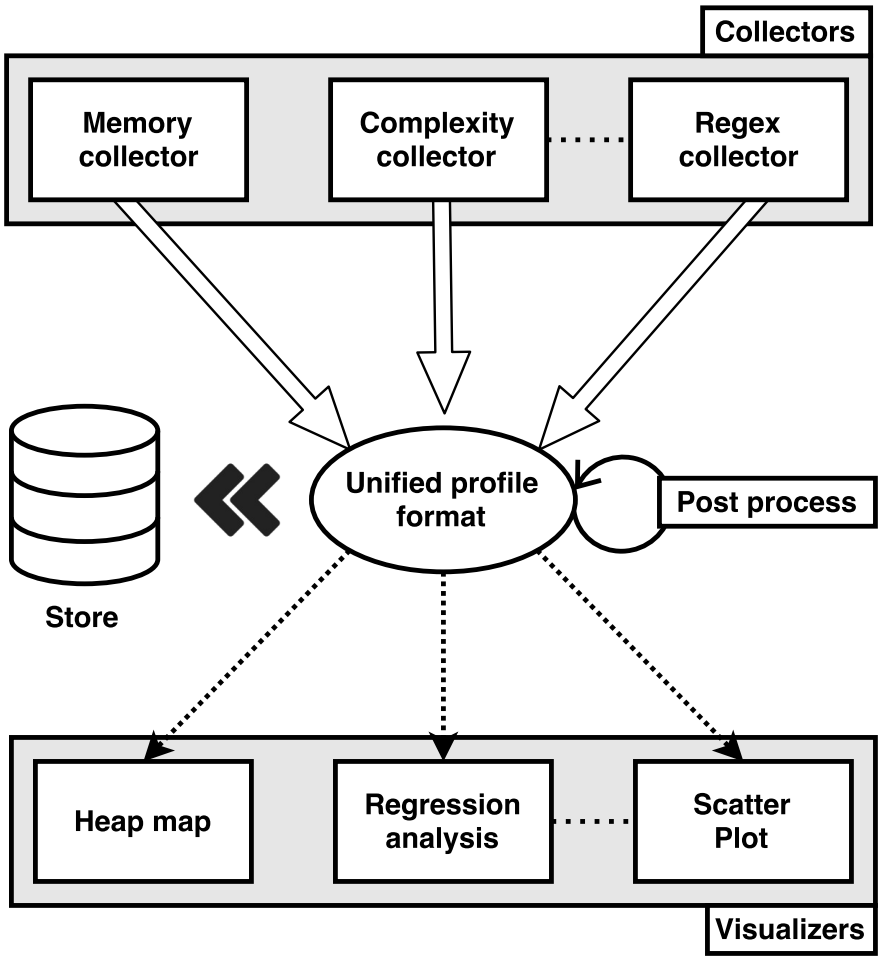
given job with specified collection (resp. postprocessing) unit and generatesj profile (potentially in raw or intermediate format). Finally the `after()` phase is run, which can further postprocess the generated profile (after the success of collection), e.g. by required filtering of data or by transforming 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`.

Before this overall process, one can run a custom set of commands by stating the key *`execute.pre_run`* key. This is mostly meant for compiling of new version or preparing other necessary requirements before actual collection.



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!"
```

From version 15.1. you can use the workload generators instead. See *List of Supported Workload Generators* for more information about supported workload generators and `generators.workload` for more information how to specify the workload generators in the configuration files.

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
```

7.4 List of Supported Workload Generators

From version 0.15.1, Perun supports the specification of workload generators, instead of raw workload values specified in `workloads`. These generators continuously generates workloads and internally Perun either merges the resources

into one single profile or gradually generates profile for each workload.

The generators are specified by *generators.workload* section. These specifications are collected through all of the configurations in the hierarchy.

You can use some basic generators specified in shared configurations called `basic_strings` (which generates strings of lengths from interval (8, 128) with increment of 8), `basic_integers` (which generates integers from interval (100, 10000), with increment of 200) or `basic_files` (which generates text files with number of lines from interval (10, 10000), with increment of 1000).

7.4.1 Generic settings

All generators can be configured using the following generic settings:

- `profile_for_each_workload`: by default this option is set to false, and then when one uses the generator to generate the workload, the collected resources will be merged into one single profile. If otherwise this option is set to true value (true, 1, yes, etc.) then Perun will generate profile for each of the generated workload.

7.4.2 Singleton Generator

Singleton Generator generates only one single value. This generator corresponds to the default behaviour of Perun, i.e. when each specified workload in *workloads* was passed to profiled program as string.

Currently be default, any string specified in *workloads*, that does not correspond to some generator specified in *generators.workload*, is converted to Singleton Generator.

The Singleton Generator can be configured by following options:

- `value`: singleton value that is passed as workload.

7.4.3 Integer Generator

Integer Generator generates the range of the integers.

The Integer Generator starts from the `min_range` workload, and continuously increments this value by `step` (by default equal to 1) until it reaches max_range (including).

The following shows the example of integer generator, which continuously generates workloads 10, 20, . . . , 90, 100:

```
generators:
  workload:
    - id: integer_generator
      type: integer
      min_range: 10
      max_range: 100
      step: 10
```

The Integer Generator can be configured by following options:

- `min_range`: the minimal integer value that shall be generated.
- `max_range`: the maximal integer value that shall be generated.
- `step`: the step (or increment) of the range.

7.4.4 String Generator

String Generator generates strings of changing length.

The String Generators starts generating random strings starting from the `min_len`, and continuously increments this length by `step_len` (by default equal to 1), until it reaches the `max_len` (including).

The following shows the example of integer generator, which continuously generates workload strings of length 1, 2, . . . , 9, 10:

```
generators:
  workload:
    - id: string_generator
      type: string
      min_len: 1
      max_len: 10
      step_len: 1
```

The String Generator can be configured by following options:

- `min_len`: the minimal length of the string that shall be generated.
- `max_len`: the maximal length of the string that shall be generated.
- `step_len`: the step (or increment) of the lengths.

7.4.5 Text File Generator

Text File Generator generates the range of random files.

The TextFile Generator generates files with random contents (lorem ipsum) starting from `min_lines`, and continuously increments this value by `step` until the number of lines in the generated file reaches `max_lines`. Each row will then either have maximal length of `max_chars` (if `randomize_rows` is set to false value), otherwise the length is randomized from the interval (`min_lines`, `max_lines`).

The following shows the example of integer generator, which continuously generates workloads 10, 20, . . . , 90, 100:

```
generators:
  workload:
    - id: textfile_generator
      type: textfile
      min_lines: 10
      max_lines: 100
      step: 10
```

The TextFile Generator can be configured by following options:

- `min_lines`: the minimal number of lines in the file that shall be generated.
- `max_lines`: the maximal number of lines in the file that shall be generated.
- `step`: the step (or increment) of the range. By default set to 1.
- `min_chars`: the minimal number of characters on one line. By default set to 5.
- `max_chars`: the maximal number of characters on one line. By default set to 80.
- `randomize_rows`: by default set to True, the rows in the file have then randomized length from interval (`min_chars`, `max_chars`). Otherwise (if set to false), the lines will always be of maximal length (`max_chars`).

CHAPTER

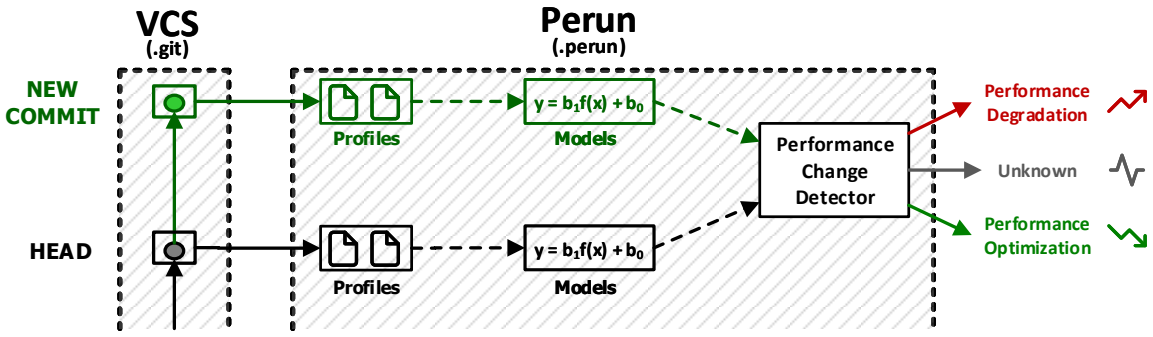
EIGHT

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

The *Best Model Order Equality* chooses 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.

- **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 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

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)

- **Detects**: *Ratio* changes; `Optimization` and `Degradation`
- **Confidence**: *None*

- **Limitations**: *None*

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.

8.2.3 Fast Check

The module contains the method for detection with using regression analysis.

This module contains method for classification the perfomance change between two profiles according to computed metrics and models from these profiles, based on the regression analysis.

8.2.4 Linear Regression

The module contains the method for detection with using linear regression.

This module contains method for classification the perfomance change between two profiles according to computed metrics and models from these profiles, based on the linear regression.

8.2.5 Polynomial Regression

The module contains the method for detection with using polynomial regression.

This module contains method for classification the perfomance change between two profiles according to computed metrics and models from these profiles, based on the polynomial regression.

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.utils.structs.**DegradationInfo**(*res*, *loc*, *fb*, *tt*, *t='-'*, *rd=0*, *ct=0*, *cr=0*, *pi=None*)

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** (*float*) – value of the confidence we have in the detected degradation

to_storage_record()

Transforms the degradation info to a storage_record

Returns string representation of the degradation as a stored record in the file

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:

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

from perun.utils.structs import DegradationInfo


def my_degradation_checker(baseline_profile, target_profile):
    """..."""
    yield DegradationInfo("...")
```

3. Next, in the `__init__.py` module register the short string for your new method as follows:

```
--- /mnt/e/phdwork/perun/git/docs/_static/templates/degradation_init.
↪py
+++ /mnt/e/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).

CHAPTER

NINE

PERFORMANCE FUZZ-TESTING

Unfortunately, in our experience, manually created test cases usually do not detect hidden performance bugs, because they do not cover all cases of inputs. The performance testing of ones application heavily depends on input workloads. So in order to have best set of input workloads, it is appropriate to adapt more advanced techniques of testing.

Fuzzing is a well-known testing technique used to find vulnerabilities in applications by sending garbled data as an input and then monitoring the application for crashes. It has been shown that even just an aggressive random testing is impressively effective at finding faults and has enjoyed great success at discovering security-critical bugs as well. Using fuzz testing, developers and testers can 'hack' their systems to detect potential security threats before attackers can. So why should not we use fuzzing to discover implementation faults affecting performance?

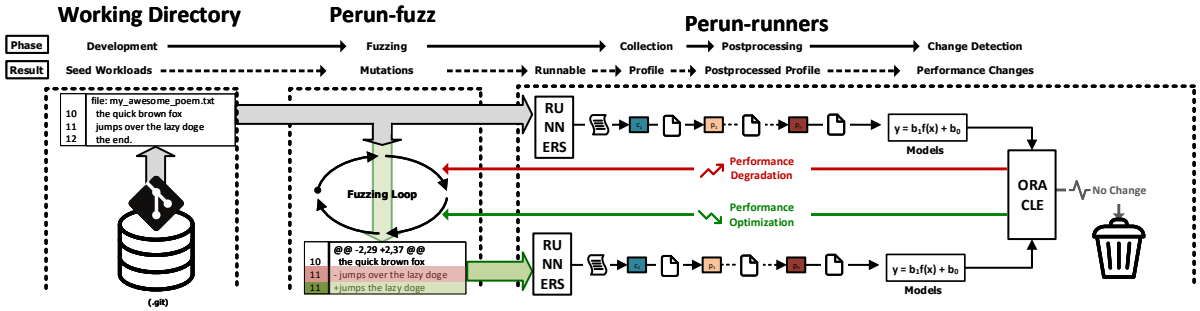
We noticed that, while there are many projects implementing fuzz testing technique, unfortunately, none of them allows to add custom mutation strategies which could be more adapted for the target program and mainly for triggering performance bugs. In Perun, we proposed a modification of fuzz testing unit that specializes for producing inputs greedy for resources. We proposed new mutation strategies inspired by causes of performance bugs found in real projects. We believe that combining performance versioning and fuzzing could raise the ratio of successfully found performance bugs early in the process.

9.1 Overview

The underdeveloped field of performance fuzz testing has inspired us to explore this field more and extend the Perun tool with fuzzing module that tries to find new workloads (or inputs) that will likely cause a change in program performance. In particular, the fuzzing mode of Perun offers:

1. **New mutation rules**: We devised new rules designed to affect performance. Our group of rules is general, and does not focus on the only one type of potential performance problem and tackles several types of input files and their associated performance issues.
2. **Classic rules**: The existing fuzzers proposed so called *classical rules*, and they have achieved great success in past, therefore we adapt the classic generally used mutation rules to our collection of rules as well.
3. **Perun-based evaluation**: We select inputs for mutation mainly according to the Perun results, instead of using classical evaluation criteria.
4. **Heuristics based on coverage testing**: The fuzzing is in general a brute-force technique, and so we do not want to test with Perun every workload, since Perun adds considerable overhead for each testing. We implement a heuristic, that first tests the coverage of the code to quickly filter out completely uninteresting workloads before evaluating them by Perun.
5. **Interpretation of mutated workloads**: We believe that after the fuzz testing, testers primarily want to know what workloads are making the troubles to application and how they differ from the original files. We propose a simple technique for visualizing the results of the fuzzing by showing the differences between input seeds and their resulting mutations.

6. **Interpretation of the fuzzing process**: Additionally to visualization of inferred workload, we also provide several graphs that illustrates the illustrates the fuzzing process itself. This allows developers to tune out regular fuzzing testings to achieve best results in best possible time.



Our solution currently modifies input workload *files* (one of the most common format of program workload) based on *mutational* approach. The feedback loop is extended with coverage information, for the purpose of increasing the efficiency and chances to find the worst-case workloads and is used as initial test for finding possible time-consuming workloads. After the initial evaluation we use Perun, to automatically detects performance changes based on the data collected within the program runtime.

For different file types (or those of similar characteristics) we use different groups of mutation methods. Hence, we apply domain-specific knowledge for certain types of files to trigger the performance change or find unique errors more quickly.

Before the actual fuzzing loop, we first determine the *performance baseline*, i.e. the expected performance of the program, to which future results (so called targets) will be compared. In initial testing we first measure code coverage (number of executed lines of code) while executing each initial seed. The median of measured coverage data is then considered as the baseline for coverage testing. Second, Perun is run to collected memory, time or trace resource records with initial seeds resulting into baseline profiles (`base_profile`). Practically *performance baseline* profiles describe the performance of the program on the given workload corpus. After the initial testing, the seeds in the corpus are considered as parents for future mutations and rated by the evaluation function.

The fuzzing loop itself starts with choosing one individual file from corpus (initial seeds). This file is then transformed into mutations. We first precollect the interesting mutations: those that increase the number of executed lines. We argue that prefiltering the results with coverage based testing is fast and can yield satisfying results. In later step, we combine these results with the performance check, which is on the other hand slower, but yields precise results

After precollecting the interesting workloads, we collect run-time resources (memory, trace, time) using Perun's collectors (see *Collectors Overview*), transforms them to so called target profiles and checks for performance changes by comparing newly generated target profile with baseline performance profile (see *Detecting Performance Changes* for more details about degradation checks). We repeat, that the intuition is, that running coverage testing is faster than collecting performance data (since it introduces certain overhead) and collecting performance data only for possibly newly covered paths could result into more interesting workloads. According to the number of gathered workloads we adapt the coverage increase ratio, with an aim to either mitigate or tighten the condition for classification a workload as an interesting one.

List of results of each testing iteration in the main loop contains successful mutations and the history of the used rules, that led to their current form. Collection of interesting workloads is limited by two parameters: the current number of program executions (specified by option `--execs-limit`) and the current number of collected files (specified by `--interesting-files-limit`). The first limit guarantees that the loop will terminate. On the other hand, if it is set to excessively high value, it would lead to a long duration of this phase, especially if the test program itself is used to run for a longer time. The latter limit ensures the loop will end in reasonable time and collects reasonable number of workloads. The combination of these limits ensures termination in reasonable time.

Note, that we can collect line coverage only in the presence of source files. In case we are supplied only with binary or script, we skip the first (and fast) testing phase and only checks for possible performance changes.

9.2 Mutation Strategies

In general, the goal of mutational strategies is to randomly modify a workload to create a new one. We propose a series of rules inspired by both existing performance bugs found in real projects, and general knowledge about used data structures, sorting algorithms, or regular expressions.

Both the types of workloads and the rules for their modification are divided into several groups: *text*, *binary* and *domain specific*. In particular, we currently support domain-specific rules for XML format based files. We identify each rule with its own label name (T stands for text, B for binary and D for domain-specific), with a brief description of what it concentrates on and the demonstration result of its application on some sample data. In case the rule is inspired by concrete bug found in real application, we list the link to the report. Collects fuzzing rules specific for text files.

perun.fuzz.methods.textfile.**change_character**()

Rule T.4: Change random character

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the quack brown ‘‘b‘‘ox jumps over the lazy dog"
- **Description**: Adaptation of classical rule for text files. Changes a random character at at random line to different character.
- **Known Issues**: none

perun.fuzz.methods.textfile.**delete_character**()

Rule T.15: remove a random character.

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the quck brown fox jumps over the lazy dog"
- **Description**: Removes a random character in random word in random line.
- **Known Issues**: none

perun.fuzz.methods.textfile.**divide_line**()

Rule T.3:

- **Input**: "<author>Gambardella, Matthew</author>"
- **Mutation**: "<author>Gambardella, Matthew</au", "thor>"
- **Description**: Divides a line by inserting newline character in random position.
- **Known Issues**: none

perun.fuzz.methods.textfile.**double_line**()

Rule T.1: Double the size of a line

- **Input**: "The quick brown fox."
- **Mutation**: "The quick brown fox.The quick brown fox."
- **Description**: This rule focuses on possible performance issues associated with long lines appearing in files. The rule doubles the selected random line in the input.
- **Known Issues**:
  1. gedit text editor (issue with too long lines)
  2. Poorly validated regexps (issue with lenghty backtracking)

perun.fuzz.methods.textfile.**duplicate_line**()

**Rule T.2: Duplicate a line **

- **Input**: “The quick brown fox.”
- **Mutation**: “The quick brown fox.”, “The quick brown fox.”
- **Description**: Extends the file vertically, by duplicating random line in the file.
- **Known Issues**: none

perun.fuzz.methods.textfile.**delete_line**()

Rule T.13: Remove random line

- **Input**: “the quick brown fox jumps over the lazy dog”
- **Mutation**: “”
- **Description**: Removes random line.
- **Known Issues**:

perun.fuzz.methods.textfile.**append_whitespace**()

Rule T.8: Append whitespaces

- **Input**: “the quick brown fox jumps over the lazy dog”
- **Mutation**: “the quick brown fox jumps over the lazy dog‘‘ ‘‘”
- **Description**: The rule appends random number of whitespaces at random line.
- **Known Issues**: none

perun.fuzz.methods.textfile.**insert_whitespace**()

Rule T.10: Insert whitespaces on a random place.

- **Input**: “the quick brown fox jumps over the lazy dog”
- **Mutation**: “The quick bro‘‘ ‘‘wn fox jumps over the lazy dog”
- **Description**: The rule inserts random number of whitespaces at random place in the random line. There are several intuitions behind this rule: (1) some trimming regular expressions can induce the excessive number of backtracking, and (2) some structures, such as hash tables, can have bad properties and lead to a singly-linked list when induced with lots of words (e.g. when one chooses wrong size of the table or bad hash-function.
- **Known Issues**: none

perun.fuzz.methods.textfile.**prepend_whitespace**()

Rule T.9: Prepend whitespaces

- **Input**: “the quick brown fox jumps over the lazy dog”
- **Mutation**: “‘‘ ‘‘The quick brown fox jumps over the lazy dog”
- **Description**: The rule prepends random number of whitespaces at random line.
- **Known Issues**:
  1. StackOverflow regular expression with quadratic number of backtrackings.

perun.fuzz.methods.textfile.**repeat_whitespace**()

Rule T.11: Repeat whitespaces.

- **Input**: “the quick brown fox jumps over the lazy dog”
- **Mutation**: “The quick brown‘‘ ‘‘ fox jumps over the lazy dog”
- **Description**: The rule repeats random number of whitespaces at random place in the random line. There intuition behind this rule is that some trimming regular expressions can induce the excessive number of backtracking.

- **Known Issues**: none

perun.fuzz.methods.textfile.**bloat_words**()

Rule T.12: Remove whitespaces.

- **Input**: "The quick brown fox."
- **Mutation**: "The quickbrown fox."
- **Description**: Removes whitespace from a random line. The intuition is to create a bigger words that might bloat the underlying structures.
- **Known Issues**: none

perun.fuzz.methods.textfile.**repeat_word**()

Rule T.5: Repeat random word of a line

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the quick brown `brown` fox jumps over the lazy dog"
- **Description**: The rule picks a random word in random line and repeats it several times. The intuition is, that there e.g. exist certain vulnerabilities, when repeated occurrences of words can either lead to faster (e.g. when the word is cached) or slower time (e.g. when in hash-table the underlying structure is degradated to list). Moreover, some algorithms, such as quick sort are forced to worst-case, when all elements are same.
- **Known Issues**: none

perun.fuzz.methods.textfile.**delete_word**()

Rule T.14: Remove random word

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the brown fox jumps over the lazy dog"
- **Description**: Removes random word in random line.
- **Known Issues**: none

perun.fuzz.methods.textfile.**sort_line**()

Rule T.6: Sort words or numbers of a line.

- **Input**: "The quick brown fox."
- **Mutation**: "brown fox quick The.
- **Description**: The intuition of this rule is to force bad behaviour, e.g. to sorting algorithm, that in some cases perform worse for sorted output, or to balanced trees, which might be unbalanced for sorted values.
- **Known Issues**: none

perun.fuzz.methods.textfile.**sort_line_in_reverse**()

Rule T.7: Sort words or numbers of a line in reverse.

- **Input**: "The quick brown fox."
- **Mutation**: "brown fox quick The.
- **Description**: The intuition of this rule is to force bad behaviour, e.g. to sorting algorithm, that in some cases perform worse for sorted output, or to balanced trees, which might be unbalanced for sorted values.
- **Known Issues**: none

In case of binary files we cannot apply specific domain knowledge nor can we be inspired by existing performance issues. Instead, we mostly adapt the classical fuzzing rules.

perun.fuzz.methods.binary.**insert_byte**()

Rule B.3: Insert random byte.

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the qui#ck brown fox jumps over the lazy dog"
- **Description**: Implementation of classical fuzzing rule.
- **Known Issues**: none

perun.fuzz.methods.binary.**remove_byte**()

Rule B.4: Remove random byte.

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the quik brown fox jumps over the lazy dog"
- **Description**: Implementation of classical fuzzing rule.
- **Known Issues**: none

perun.fuzz.methods.binary.**swap_byte**()

Rule B.5: Swap random bytes.

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the quock brown fix jumps over the lazy dog"
- **Description**: Implementation of classical fuzzing rule. Picks two random lines and two random bytes in the line and swaps them.
- **Known Issues**: none

perun.fuzz.methods.binary.**insert_zero_byte**()

Rule B.2: Insert random zero byte.

- **Input**: This is C string. You are gonna love it.
- **Mutation**: This is string.'''' You are gonna love it.
- **Description**: The rule inserts random zero byte \ in the string. The intuition is to target the C language application, that process the strings as zero-terminated string of bytes.
- **Known Issues**: none

perun.fuzz.methods.binary.**remove_zero_byte**()

Rule B.1: Remove random zero byte

- **Input**: This is C string.You are gonna love it.
- **Mutation**: This is string. You are gonna love it.
- **Description**: The rule removes random zero byte \ in the string. The intuition is to target the C language application, that process the strings as zero-terminated string of bytes. Removing the zero byte could lead to program non-termination, or at least crashing when reading the whole memory.
- **Known Issues**: none

perun.fuzz.methods.binary.**flip_bit**()

Rule B.6: Flip random bit.

- **Input**: "the quick brown fox jumps over the lazy dog"
- **Mutation**: "the quack brown fox jumps over the lazy dog"
- **Description**: Implementation of classical fuzzing rule.

- **Known Issues**: none

Exploiting more domain-specific knowledge about the workload we devised specific rules for concrete formats. We propose rules for removing tags, attributes, names or values of attributes used in XML based files (i.e. `.xml`, `.svg`, `.xhtml`, `.xul`). For example, we can assume a situation, when fuzzer removes closing tag, which will increase the nesting. Then a recursively implemented parser will fail to find one or more of closing brackets (representing recursion stop condition) and may hit a stack overflow error.

`perun.fuzz.methods.xml.`**`remove_attribute_value`**`()`

Rule D.3: Removed attribute value.

- **Input**: <book id="bk106" pages="457">
- **Mutation**: <book id="bk106" pages="">
- **Description**: Removes random value of the attribute in the random line and tag.
- **Known Issues**: none

`perun.fuzz.methods.xml.`**`remove_attribute_name`**`()`

Rule D.2: Remove attribute name.

- **Input**: <book id="bk106" pages="457">
- **Mutation**: <book id="bk106" "457">
- **Description**: Removes name of the attribute in random tag in the random line.
- **Known Issues**: none

`perun.fuzz.methods.xml.`**`remove_attribute`**`()`

Rule D.1: Remove an attribute.

- **Input**: <book id="bk106" pages="457">
- **Mutation**: <book id="bk106">
- **Description**: Selects random tag and removes a random attribute.
- **Known Issues**: none

`perun.fuzz.methods.xml.`**`remove_tag`**`()`

**Rule D.: **

- **Input**: <book id="bk106" pages="457">
- **Mutation**:
- **Description**: Removes a random tag.
- **Known Issues**: none

We further offer the possibility of adding custom rules. For adding the rules to a mutation strategy set, you can launch the fuzzer with a special file in YAML file format containing the description of applied rules using the `--regex-rules` option. Each rule is represented as an associative array in a form *key: value*, where both are regular expressions but *key* is a pattern which should be replaced, and value* is the replacement.

```
Back: Front
del: add
remove: create
([0-9]{6}),([0-9]{2}): \\1.\\2
(\\w+)=(\\w+): \\2=\\1
```

Additionally, one can extend the existing rules by modifying files `binary.py`, `textfile.py` or `xml.py` in the `methods` package. Further, it is necessary to modify the script `filetype.py`, which is responsible for selecting

the rules. To add, for example, specific rules for JSON file type, one just has to create a new script, e.g. `json.py`, and modify the rules selection. Note that every rule should contain a brief description, which will be displayed after fuzzing.

9.3 Passing Input Sample

Workloads can be passed to fuzzer as an arbitrary mix of files or directories using the `--input-sample` option. Directories are then iteratively walked for all files with reading permissions. Optionally files can be filtered using option `--workloads-filter`: a user specified regular expression that file names must match. E.g one can fuzz with XML files by setting expression `--workloads-filter="^.*.xml$"`. Or if one wants to skip all the files with the name containing string "error" one can use `--workloads-filter="^((?!error).)*$"`. Note that the fuzzer should always be launched with just one type of initial files even if the target application supports more types, since we tune the rules according to workload file format.

9.4 Selecting Mutation Methods

We select corresponding mutation strategies based on the first loaded workload file. Basically, if the file is a binary, all the rules specific to binaries are added to the set of rules, otherwise we add all the basic text rules. We further analyse the mime type of a file and if it is supported by the fuzzer, we add to the set of rules mime-specific rules as well as any user-defined rules.

We argue the advantage of fuzzing with one file type rests in its code covering feature. To be more precise, we are not observing at the overall percentage of code coverage, but how many lines of code has been executed in total during the run, with an aim to maximise it. Consider an application that extracts meta-data from different media files, such as WAV, JPEG, PNG, etc. If a PNG image file is used as a seed to this application, only the parts related to PNG files will be tested. Then testing with WAV will cause, that completely different parts of the program will be executed, hence total executed code lines of these two runs cannot compare with each other because reaching higher line coverage with WAV files would lead to preferring them for fuzzing, and PNG files would be neglected. Moreover, we are aware that this strategy may miss some performance bugs. Fuzzing multiple mime-types is current feature work.

9.5 Initial Testing

The newly mutated results have to be compared against some expected behaviour, performance or value: so called baseline results (i.e. results and measurements of workload corpus). Hence, initial seeds become test cases and they are used to collect performance baselines. By default, our initial program testing (as well as testing within the fuzzing loop) interleaves two phases described in more details below: coverage and performance-guided testing.

In `perun-fuzz`, we use gcov tool to measure the code coverage. Note that the program has to be build for coverage analysis with GNU Compiler Collection (GCC) with the option `--coverage` (or alternatively a pair of options `-fprofile-arcs -ftest-coverage`). The resulting file with the extension `.gcno` then contains the information about basic block graphs and assigns source line numbers to blocks. If we execute the target application a separate `.gcda` files are then created for each object file in the project. These files contain arc transition counts, value profile counts, and additional summary information gcov.

Total count of executed code lines through all source files represents the baseline coverage (and partly also a performance) indicator. An increase of the value means that more instructions have been executed (for example, some loop has been repeated more times) so we hope that performance degradation was likely triggered as well. Note that the limitation of this approach is that it does not track uniquely covered paths, which could trigger performance change as well. Support of more precise coverage metrics is a future work.

First the target program is executed with all workloads from corpus. After each execution, `.gcda` files are filled with coverage information, which Gcov tool parses and generates output files. We parse coverage data from the output `.gcov` file, sum up line executions, compare with the current maximum, update the maximum if new coverage is greater and iterate again. It follows that base coverage is the maximum count of executed lines reached during testing with seeds.

While coverage-based testing within fuzzing can give us fast feedback, it does not serve as an accurate performance indicator. We hence want to exploit results from Perun. We run the target application with a given workload, collect performance data about the run (such as runtime or consumed memory) and store them as a persistent profile (i.e. the set of performance records). Again, we will need a performance baseline, which will be compared with newly generated mutations. Profiles measured on fuzzed workloads (so called *target profiles*) are then compared with a profile describing the performance of the program on the initial corpus (so called*baseline profiles*). In order to compare the pair of baseline and target profiles, we use sets of calculated regression models, which represents the performance using mathematical functions computed by the least-squares method. We then use the Perun internal degradation methods (see *Detecting Performance Changes*.

9.6 Evaluation of Mutations

Initially, the workload corpus is filled with seeds (given by user), which will be parents to newly generated mutations (we can also call these seeds *parent workloads*). In the main loop, we extend this corpus with successful mutations which in retrospect become *parent workloads* too. The success of every workload is represented by the *fitness score*: a numeric value indicating workload's point rating. The better rating of workload leads either to better code coverage (and possibly new explored paths or iterations) or to newly found performance changes. We calculate the total score by the following evaluation function:

$score_{workload} = icovr_{\text{workload}} * (1 + pcr_{\text{workload}})$.

Increase coverage rate (icovr): This value indicates how much coverage will change if we run the program with the workload, compared to the base coverage measured for initial corpus. Basically, it is a ratio between coverage measured with the mutated workload and the base coverage:

$icovr_{\text{workload}} = cov_{\text{workload}}/cov_{\text{base}}$.

Performance change rate (pcr): In general, we compare the newly created profile with the baseline profile and the result is a list of located performance changes (namely *degradations*, *optimisations* and *no changes*). Performance change rate is then computed as ratio number of degradations in the result list:

$pcr_{\text{workload}} = \text{cnt}(degradation, result)/\text{len}(result)$

This value plays a large role in the overall ranking of workload, because it is based on the real data collected from the run. And so workloads that report performance degradations and not just increases coverage have better ranking. The computation of pcr_{workload} could further be extended by the rate of degradations, i.e. if two workloads found the same number of degradations, the workload which contains more serious change would be ranked better. Optimisations of ranking algorithm is another future work. This evaluation serves for informed candidate selection for fuzzing from the parents.

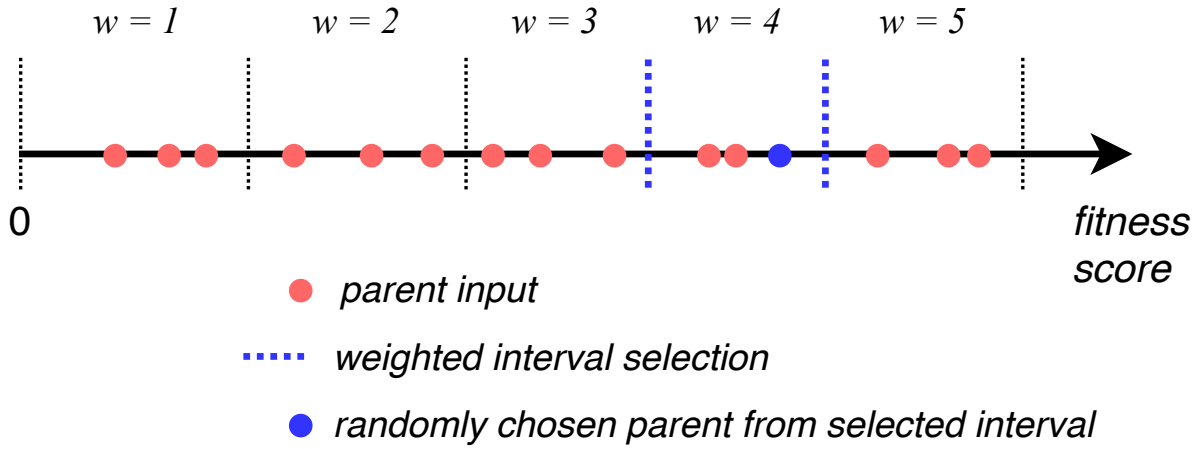
9.7 Fuzzing Loop

We can catch SIGINT signal to terminate the fuzzing when one decides to quit earlier. Fuzz unit of Perun catches this signal, however, other Perun units (collectors, postprocessors) have not implemented handlers for interruption signal, hence it is not recommended to interrupt during performance testing, but only in the coverage-guided testing phase. In the following, we will describe selected aspects of the main loop of the whole fuzzing process.

At the beginning of every iteration we first select the workloads from parents which will be further mutated. All parents are kept sorted by their scores, and the selection for mutation consists of dividing the seeds into five intervals

such that the seeds with similar value are grouped together. In our experience, five intervals seem to be appropriate because with fewer intervals parents are in too big and inappropriate groups and in the case of more intervals, parents with similar score are pointlessly scattered.

First, we assign a weight to each interval using linear distribution. Then we perform a weighted random choice of interval. Finally, we randomly choose a parent from this interval, whereas differences between parent's scores in the same interval are not very notable. The intuition behind this strategy is to select the workload for mutation from the best rated parents. From our experience, selecting only the best rated parent in every iteration does not lead to a better results, and other parents are naturally ignored. Hence we do selection from all the parents, but the parent with better score has a greater chance to be selected.



In each iteration of fuzzing we generate new workloads. However, we first determine how many new mutation (N) to generate by rule f in the current iteration of fuzzing loop. If N is too big and we generate mutations for each rule f from the set of rules, the corpus will bloat. On the other hand, if N is too low, we might not trigger any change at all, as we will not prefer successful rules more. Instead we propose to dynamically calculate the value of N according to the statistics of fuzzing rules during the process. Statistical value of rule f is a function:

$stats_f = (degs_f + icovr_f)$

where $degs_f$ represents the number of detected degradations by applying the rule f, and $icovr_f$ stands for how many times the coverage was increased by applying rule f. Fuzzer then calculates the number of new mutations for every rule to be applied in four possible ways using `--mutations-per-rule` option:

1. `--mutations-per-rule=unitary`. The case when $N = 1$, the fuzzer will generate one mutation per each rule. This is a simple heuristic without the usage of statistical data and where all the rules are equivalent.
2. `--mutations-per-rule=proportional`. The case when $N = min(stats_f + 1, FLPR)$, the fuzzer will generate mutations proportionally to the statistical value of function (i.e. $stats_f$). More mutation workloads are generated for more successful rules. In case the rule f has not caused any change in coverage or performance (i.e. $stat_f = 0$) yet, the function will ensure the same result as in the first strategy. File Limit Per Rule (FLPR) serves to limit the maximum number of created mutations per rule and is set to value 100.
3. `--mutations-per-rule=probabilistic`. Heuristic that depends on the total number of degradation or coverage increases ($total$). The ratio between $stats_f$ and $total$ determines the probability $prob_f$, i.e. the probability whether the rule f should be applied, as follows:

$$prob_f = \begin{cases} 1 & \text{if } total = 0 \\ 0.1 & \text{if } stats_f/total < 0.1 \\ stats_f/total & \text{otherwise} \end{cases}$$

and we choose N as:

$$N = \begin{cases} 1 & \text{if } random <= prob_f \\ 0 & \text{otherwise} \end{cases}$$

Until some change in coverage or performance occurs, (i.e. while $total = 0$), one new workload is generated by each rule. After some iterations, more successful rules have higher probability, and so they are applied more often. On contrary rules with a poor ratio will be highly ignored. However, since they still may trigger some changes we round them to the probability of 10%.

4. `--mutations-per-rule=mixed`. The last heuristic is a modified third strategy combined with the second one. When the probability is high enough that the rule should be applied, the amount of generated workloads is appropriate to the statistical value. Probability $prob_f$ is calculated equally, but the equation for choosing N is modified to:

$$N = \begin{cases} min(stats_f + 1, FLPR) & \text{if } random <= prob_f \\ 0 & \text{otherwise} \end{cases}$$

Our fuzzer uses by default the last heuristic, `--mutations-per-rule=mixed`, because in our experience it guarantees that it will generate enough new workloads each iteration and will as well filter out unsuccessful rules without totally discarding them. In case that target program is prone to workload change and the user wants better interleaving of testing phases, it is recommended to use the third method (`--mutations-per-rule=probabilistic`) because the maximum number of all created mutations in one iteration is limited by the number of selected mutation rules.

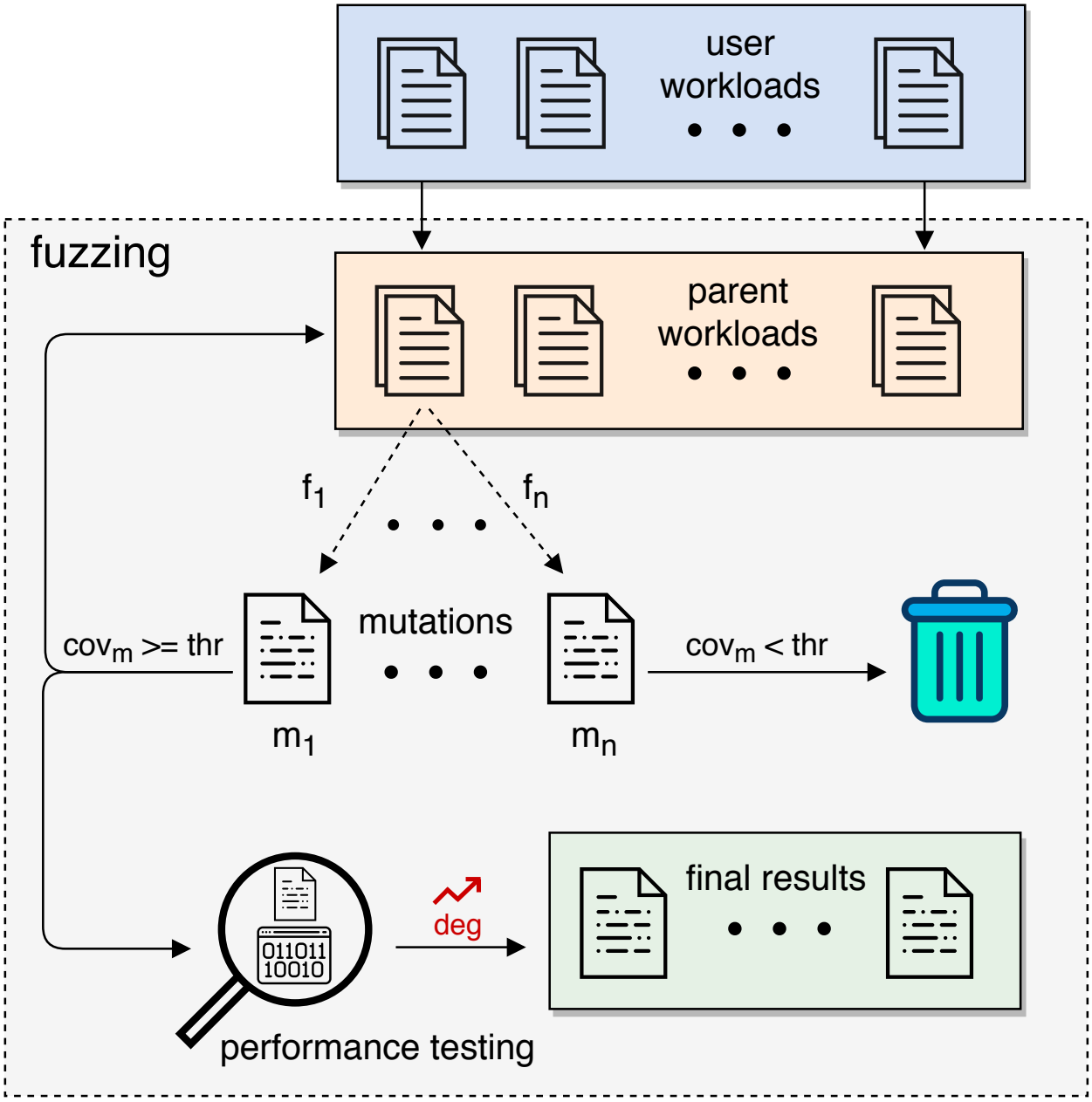
The threshold for discarding mutations is multiple of base coverage, set to 1.5 by default. The threshold can be changed by specifying the `--coverage-increase-rate`. A mutation is classified as an interesting workload in case two criteria are met:

:math:‘cov_{mut} > cov_{threshold} And cov_{mut} > cov_{parent} ‘

i.e. it has to exceed the given threshold and achieve a higher number of executed lines than its predecessor.

In addition, the constant which multiplies the base coverage (and thus determines the threshold) changes dynamically during fuzzing. In case it is problematic to reach the specified coverage threshold, the value of the constant decreases and thus gives more chance for further mutations to succeed. Vice versa, if the mutations have no problem to exceed the threshold, the value of the constant is probably too low, and hence we increase it.

During the testing, fuzzed workload can cause that target program terminates with an error (e.g. SIGSEGV, SIGBUS, SIGILL, ...) or it can be terminated by timeout. Even though we are not primarily focused on faults, they can be interesting for us as well because an incorrect internal program state can contain some degradation and in case of error, handlers can also contain degradation.



9.8 Interpretation of Fuzzing Results

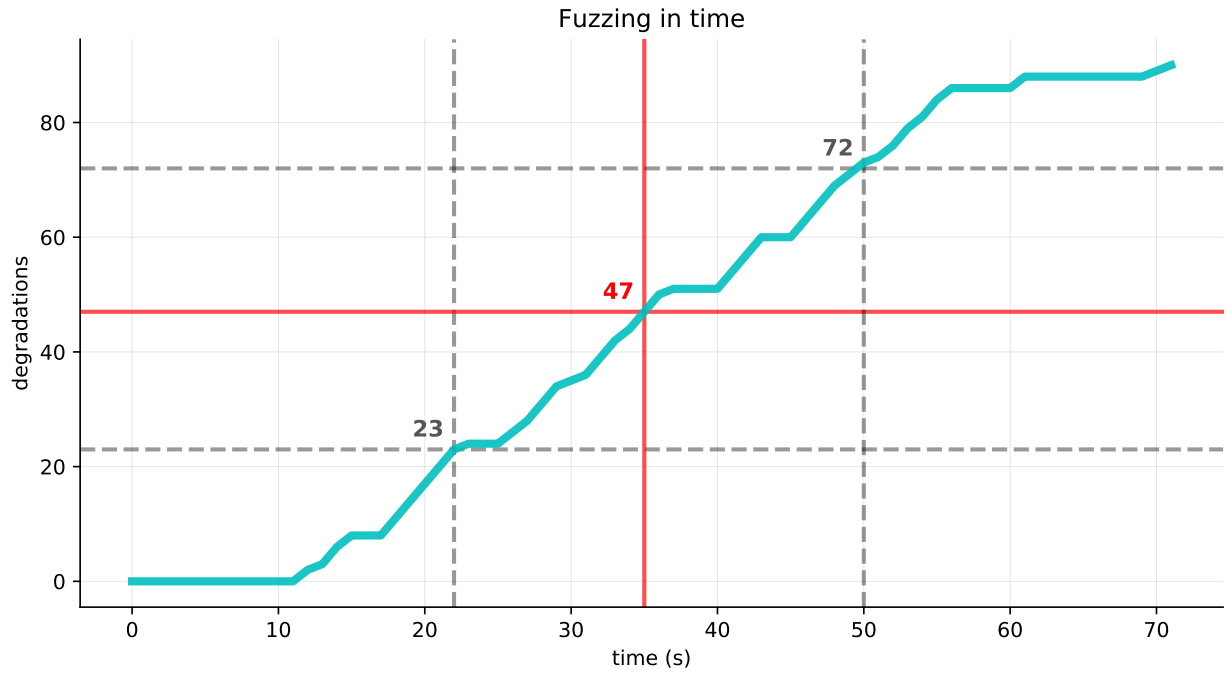
The result of the fuzzing is illustrated by the following directory structure.

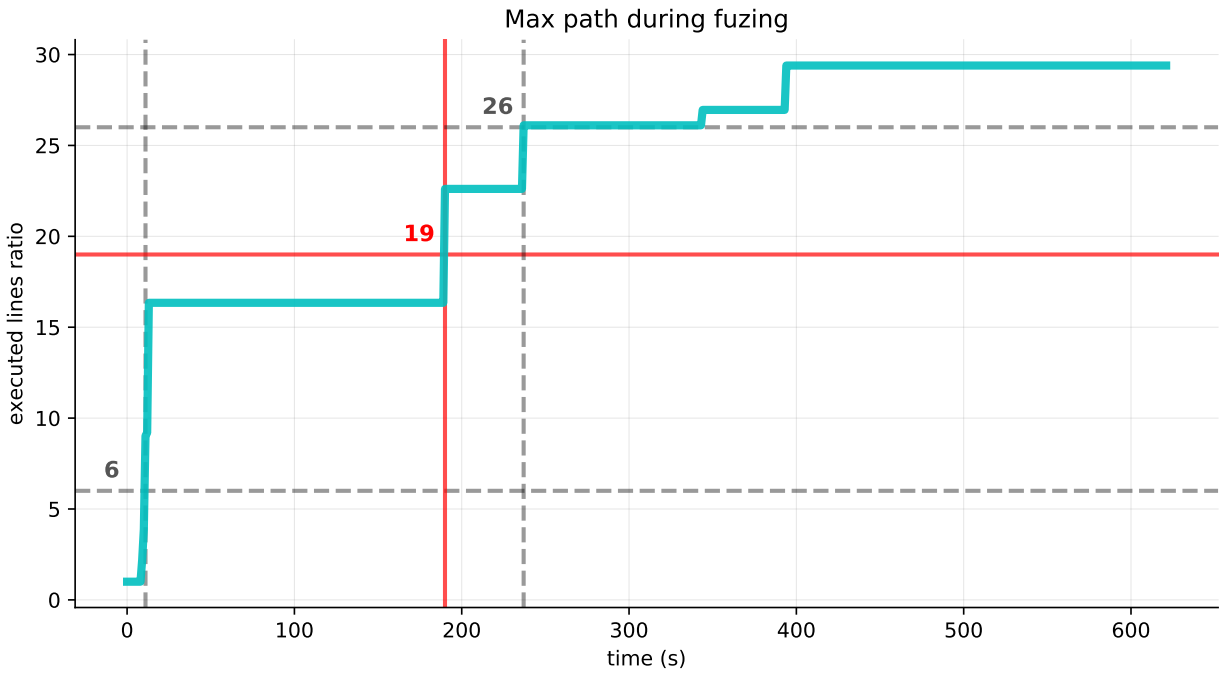
```
output_dir/
  |--- diffs/
    |--- medium_words-02000b239d024dbe933684b6c740512e-diff.html
    |--- medium_words-389d4162ad6641d187dc405000b8d50a-diff.html
    |--- medium_words-39b5d7aa55fd404aa4d31422c6513e2c-diff.html
  |--- faults/
    |--- medium_words-389d4162ad6641d187dc405000b8d50a.txt
  |--- graphs/
```

```
    |--- coverage_ts.pdf
    |--- degradations_ts.pdf
  |--- hangs
    |--- medium_words-39b5d7aa55fd404aa4d31422c6513e2c.txt
  |--- logs
    |--- coverage_plot_data.txt
    |--- degradation_plot_data.txt
    |--- results_data.txt
  |--- medium_words-02000b239d024dbe933684b6c740512e.txt
```

The results will be saved to the directory specified by `--output-dir` option. The fuzzing generates three kinds of mutations: those resulting into degradations (stored in `output_dir`), those resulting into errors or faults (stored in `output_dir/faults`), and those terminated by timeout (stored in `output_dir/hangs`). The fuzzing alos generates two time series graphs in `output_dir/graphs`), which will be described later. At last in `output_dir/diffs` are stored differences between individual mutations and their parents.

The **time series** graphs show the number of found mutations causing degradation and the maximum recorded number of lines executed per one run. From these graphs, one can e.g. read the time needed to achieve sufficient results and estimate orientation time for future testing. In both graphs are denoted three statistically significant values: first quartile, second quartile (median) and third quartile from the y-axis values. The intention is to illustrate at what point in time we have achieved the individual portion of the result. The usage of time series graphs is meant to tune the properties and options of the fuzzing process.





Besides visualisation, we create **diff file** for every output file. It shows the differences between files and the original seed, from which the file was created by mutation. The file is in HTML format, and the differences are color-coded for better orientation.

CHAPTER

TEN

EXAMPLES

In the following we briefly explore several performance issues found in real projects and a group of regular expressions that have been confirmed as harmful. All the tests ran on a reference machine Lenovo G580 using 4 cores processor Intel Core i3-3110M with maximum frequency 2.40GHz, 4GiB memory, and Ubuntu 18.04.2 LTS operating system.

10.1 Regular Expression Denial of Service (ReDoS).

In this case study, we analysed artificial programs which use `std::regex_search` with regular expressions inspired by existing reported ReDoS attacks (see e.g. redos). In nutshell, ReDoS is an attack based on algorithmic complexity where regular expression are forced to take long time to evaluate, mostly because of backtracking algorithm, and leads to the denial of service.

10.1.1 StackOverflow trim regex.

The first experiment focuses on the regular expression that caused an outage of StackOverflow in July, 2016. We constructed an artificial program that reads every line and search for match with the regular expression. We used simple source code in C performing parallel grep as an initial seed, written in 150 lines. With only two tests, we could force the vulnerability, as we show in the Table.

	size [B]	runtime [s]	executed LOC ratio	lines	whitespaces
$seed$	3535	0.096	1.00	150	306
$worst\text{-}case_1$	5000	**1.566**	24.32	5	4881
$worst\text{-}case_2$	10000	**2.611**	41.38	17	9603

The following shows the used mutation rules for each mutation:

	used mutation rules
$worst\text{-}case_1$	[T.10, T.10, T.10, T.10]
$worst\text{-}case_2$	[T.10, T.10, T.10, T.10, T.10]

10.1.2 Email validation regex.

This regular expression is part of the public RegExLib library and is marked as malicious and triggering ReDoS. We constructed a program that takes an email address from a file and tries to find a match with this regular expression. As an initial seed we used a file containing valid email address `spse1po@gmail.com`. We ran two tests, in the first case with an email that must contain the same count of characters as the seed, and in the second case it can contain twice the size.

	size [B]	runtime [s]	executed LOC ratio
$seed$	18	0.016	1.00
$worst\text{-}case_1$	18	0.176	70.83
$worst\text{-}case_2$	25	**10.098**	4470.72
$worst\text{-}case_{2hang}$	36	**>5 hours**	∞

Two rules, namely removing random character and extending a size of line, were mostly encouraged in the generation of the presented workloads.

	used mutation rules
$worst\text{-}case_1$	[T.15, T.8, T.15, T.1]
$worst\text{-}case_2$	[T.15, T.15, T.1]
$worst\text{-}case_{2hang}$	[T.15, T.15, T.1]

In the following we list the most greedy workloads from each testing and their **content**:

- $worst\text{-}case_1$: `spse1pogailcspse1p`
- $worst\text{-}case_2$: `spse1poailcospse1poailco`
- $worst\text{-}case_{2hang}$: `spse1poailcospse1poailcospse1poailco`

10.1.3 Java Classname validation regex.

This vulnerable regular expression for validation of Java class names appeared in owasp Validation Regex Repository. The testing program was similar to the previous one: ti reads a class name from a file and tries to find a match with this regular expression. Initial file had one line with string `myAwesomeClassName`. To avoid the large lines, first we set a size limit for mutations to the size of the initial seed (19 bytes), then to double and finally to quadruple of the size.

	size [B]	runtime [s]	executed LOC ratio
$seed$	19	0.005	1.00
$worst\text{-}case_1$	19	0.016	14.31
$worst\text{-}case_2$	36	1.587	2383.99
$worst\text{-}case_3$	78	**3.344**	5056.67
$worst\text{-}case_{3hang}$	78	∞	∞

We detected two orders of magnitude degradation within run of program with the worst-case from the last test case ($worst\text{-}case_3$). The fuzzer generates and stores another 26 files that was classified as hangs. By additional testing we found the $worst\text{-}case_{3hang}$ workload which had enormous impact on program performance, and program did not terminate even after 13 hours lasting run.

	used mutation rules
$worst\text{-}case_1$	[T.8, T.15, T.8, T.15, T.15, T.1, T.12, T.8, T.1]
$worst\text{-}case_2$	[T.8, T.15, T.15, T.2, T.8, T.15]
$worst\text{-}case_3$	[T.8, T.15, T.1, T.4, T.2]
$worst\text{-}case_{3hang}$	[T.8, T.15, T.1, T.15, T.2]

In Table above, we list the rules in order they was applied on the initial seeds and created malicious workloads. Removing characters together with data duplicating, appending whitespaces and other rules collaborated on generation of the worst-case mutations for this case study.

We again list the **content** of generated mutations:

- $worst\text{-}case_1$: `mywesomelassamemywm`
- $worst\text{-}case_2$: `mywesomelassamemywesomelassam`
- $worst\text{-}case_3$: `ssammyAwesomelassammyAweiomelassaVmyAwes×melassammmyAwesomelassammyAweomel`
- $worst\text{-}case_{3hang}$: `laalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaalaal`

10.2 Hash Collisions

In other experiment we analysed a simple word frequency counting program, which uses hash table with a fixed number of buckets (12289 exactly) and the maximum length of the word limited to 127. The distribution of the words in the table is ensured by the hash function. It computes a hash, which is then used as an index to the table. Java 1.1 string library used a hash function that only examined 8-9 evenly spaced characters, which then could result into collisions for long strings. We have implemented this behaviour into an artificial program. The likely intention of the developers was to save the function from going through the whole string if it is longer. Therefore, for fuzzing, we initially generated a seed with 10000 words of 20 characters and started fuzzing. To compare the results we chose the DJB hash function, as one of the most efficient hash functions.

	size [kB]	**runtime [ms]**	**LOC ratio**	**runtime [ms]**	
$seed$	210	26	1.0	13	1.0
$worst\text{-}case_1$	458		3.48		2.19
$worst\text{-}case_2$	979		7.88		4.12

After only 10 minutes of fuzzing each test case was able to find interesting mutations. We then compared the run by replacing the hash function in early Java version with DJB hash function, which computes hash from every character of a string. Table shows, that worst-case workloads have much more impact on performance of the hash table and less stable times using Java hash function, compared to DJB. With such a simple fuzz testing developers could avoid similar implementation bugs.

	used mutation rules
$worst\text{-}case_1$	[T.2, T.3, T.15, T.15, T.11, T.15]
$worst\text{-}case_2$	[T.2, T.3, T.4, T.15, T.9, T.4, T.2, T.3, T.15, T.15]

Above we show the sequence of mutation rules that transformed the seed into worst-case workloads. In this experiment the rules that duplicates data (T.2), increases number of lines (T.3), changes and removes random characters (T.4 and T.15) were the most frequent.

10.3 Fuzz-testing CLI

10.3.1 perun fuzz

Performs fuzzing for the specified command according to the initial sample of workload.

```
perun fuzz [OPTIONS]
```

Options

-b, --cmd <cmd>
The command which will be fuzzed. [required]

-a, --args <args>
Arguments for the fuzzed command.

-w, --input-sample <input_sample>
Initial sample of workloads, that will be source of the fuzzing. [required]

-c, --collector <collector>
Collector that will be used to collect performance data.

-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

-m, --minor-version <minor_version_list>
Specifies the head minor version, for which the fuzzing will be performed.

-wf, --workloads-filter <workloads_filter>
Regular expression for filtering the workloads.

-s, --source-path <source_path>
The path to the directory of the project source files.

-g, --gcno-path <gcno_path>
The path to the directory where .gcno files are stored.

-o, --output-dir <output_dir>
The path to the directory where generated outputs will be stored. [required]

-t, --timeout <timeout>
Time limit for fuzzing (in seconds). Default value is 1800s.

-h, --hang-timeout <hang_timeout>
The time limit before input is classified as a hang (in seconds). Default value is 30s.

-N, --max <max>
The maximum size limit of the generated input file. Value should be larger than any of the initial workload, otherwise it will be adjusted

-mg, --max-size-gain <max_size_gain>
Max size expressed by gain. Using this option, max size of generated input file will be set to (size of the largest workload + value).Default value is 1 000 000 B = 1MB.

-mp, --max-size-ratio <max_size_ratio>
Max size expressed by percentage. Using this option, max size of generated input file will be set to (size of the largest workload * value). E.g. 1.5, max size=largest workload size * 1.5

-e, --exec-limit <exec_limit>
Defines maximum number executions while gathering interesting inputs.

-l, --interesting-files-limit <interesting_files_limit>
Defines minimum number of gathered interesting inputs before perun testing.

-cr, --coverage-increase-rate <coverage_increase_rate>
Represents threshold of coverage increase against base coverage. E.g 1.5, base coverage = 100 000, so threshold = 150 000.

-mpr, --mutations-per-rule <mutations_per_rule>
Strategy which determines how many mutations will be generated by certain fuzzing rule in one iteration: unitary, proportional, probabilistic, mixed

-r, --regex-rules <regex_rules>
Option for adding custom rules specified by regular expressions, written in YAML format file.

-np, --no-plotting
Avoiding sometimes lengthy plotting of graphs.

CHAPTER

ELEVEN

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.

11.1 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
```

11.2 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.shortlog
`[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-regression-analysis`.

`%<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).

format.sort_profiles_by
`[recursive]` Specifies which key of the profile will be used for sorting the output of the `perun status` commands. Can be one of the following attributes specified by the class attribute `ProfileInfo.valid_attributes`:

execute
Groups various list of commands, that can be executed before specific phases. Currently this contains only `pre_run` phase, which is executed before any collection of the data. This is mainly meant to execute compiling of the binaries and other stuff to ease the development. Note that these commands are executed without shell, but any risks of commands executed by these commands fall entirely into the user hands and we have no responsibility for them.

All of these list are as follows:

```
execute:
  pre_run:
    - echo "Running the code again"
    - make
    - make install
```

The list of commands above first outputs some text into the standard output, then it runs the makefile to compile the collected binary and then installs it.

execute.pre_run
`[local-only]]` Runs the code before the collection of the data. This is meant to prepare the binaries and other settings for the actual collection of the new data.

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*

profiles
Groups various option specific for profiles, such as strategies for adding or generating profiles

profiles.register_after_run
If the key is set to a true value (can be 1, true, True, yes, etc.), then after newly generated profile (e.g. by running `perun run matrix`) is automatically registered in the appropriate minor version index.

degradation
Speficies the list of strategies and how they are applied when checked for degradation in methods.

degradation.collect_before_check
`[recursive]` If set to true, then before checking profiles of two minor versions, we run the collection for job matrix to collect fresh or unexisting profiles. By default, the output of this phase is discarded into a `devnull`. This behaviour can be changed by setting the *degradation.log_collect*.

degradation.log_collect
: [recursive] If both *degradation.log_collect* and *degradation.collect_before_check* are set to true, then the precollect phase will be saved into a log of form %minor_version$-precollect.log. Otherwise, the output will be stashed into a black hole (i.e. devnull).

degradation.apply
: [recursive] Specifies which strategies are picked for application, if more than one strategy satisfies the specified constraints. If the key is set to first, then first strategy from the ordered list of *degradation.strategies* is applied; otherwise if the key is set to all, then all of the strategies from the ordered list are applied.

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
```

generators.workload
: [gathered] Specifies generators of the workload. Each workload has to be specified by its id and type, which corresponds to the name of the generator (currently we support only Integer generator, that generates the range of values). Further you can specify rest of the params, where each workload generator has different parameters. The specification can be as follows:

```
generators:
  workload:
    - id: gen1
      type: integer
      profile_for_each_workload: True
    - id: gen2
      type: integer
      min_range: 10
      max_range: 100
      step: 10
```

  This specifies two integer workload generators gen1 and gen2. The first uses the default range, while the latter specifies the range 10, 20, . . . , 100. If profile_for_each_workload is set to true value (true, yes, etc.), then isolate profile will be generated for each collected workload. Otherwise the resulting profiles are merged into the one profile, and each resources has additional key called "workload", that allows using *Regression Analysis* of amount depending on the workload.

  For more details about supported generators refer to *List of Supported Workload Generators*.

11.3 Predefined Configuration Templates

Internally local configuration files are specified w.r.t a Jinja2 template.

This template can further be augmented by named sets of predefined configuration as follows:

1. **user** configuration is meant for beginner users, that have no experience with Perun and have not read the documentation thoroughly. This contains a basic preconfiguration that should be applicable for most of the projects—data are collected by *Time Collector* and are automatically registered in the Perun after successful run. The performance is checked using the *Average Amount Threshold*. Missing profiling info will be looked up automatically.

2. **developer** configuration is meant for advanced users, that have some understanding of profiling and/or Perun. Fair amount of options are up to the user, such as the collection of the data and the commands that will be profiled.

3. **master** configuration is meant for experienced users. The configuration will be mostly empty.

The actually set options are specified in the following table. When the option is not set (signaled by – symbol) we output in the configuration table only a commented-out hint.

	user	**developer**	**master**
cmds	auto lookup	–	–
args	–	–	–
workloads	auto lookup	–	–
collectors	*Time Collector*	–	–
degradation.strategies	*Average Amount Threshold*	*Average Amount Threshold*	–
degradation.collect_before_check	true	true	–
degradation.log_collect	true	true	–
execute.pre_run	make	make	–
profiles.register_after_run	true	–	–
format.output_profile_template	%collector%-of-%cmd%-%workload%-%date%	–	–

In **user** configuration, we try to lookup the actual commands and workloads for profiling purpose. Currently for candidate executables we look within a subfolders named `build`, `_build` or `dist` and check if we find any executables. Each found executable is then registered as profiled command. For workloads we look for any file (without restrictions), and we restrict ourselves to subfolders with names such as `workload`, `workloads`, `examples` or `payloads`. Each compatible file is then registered as workload.

Currently the templates are set by `-t` option of `perun init` command (see *Perun Commands* for details on `perun init`). By default **master** configuration is used.

11.4 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`).

CHAPTER

TWELVE

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`*.

12.1 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*.

%source%: Original source of the profile. This corresponds to the name of the generated profile and the original path.

By default the profiles are sorted according to the timestamp. The sort order can be modified by setting either the *format.sort_profiles_by* or the *Command Line Interface* option `--sort-by` to a valid profile information attribute. Setting the command line option `--sort-by` has higher priority than the key set in the *format.sort_profiles_by*.

12.2 Customizing Logs

The output of `perun log --short` is defined w.r.t. formatting string specified in configuration in *format.shortlog* 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.

%changes%: Lists a short string of overall found changes for a given minor version. Found degradations are represented by red `-`, while found optimizations are represented by green `+`.

%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.

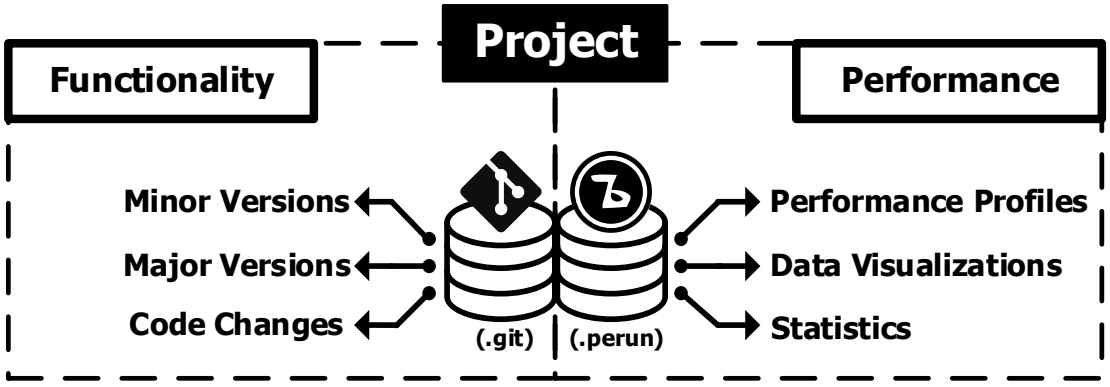
Specifying `num` in the selected tags will shorten the displayed identification to `num` characters only. In case the specified `num` is smaller then the length of the attribute name, then the shortening will be limited to the lenght of the attribute name.

CHAPTER

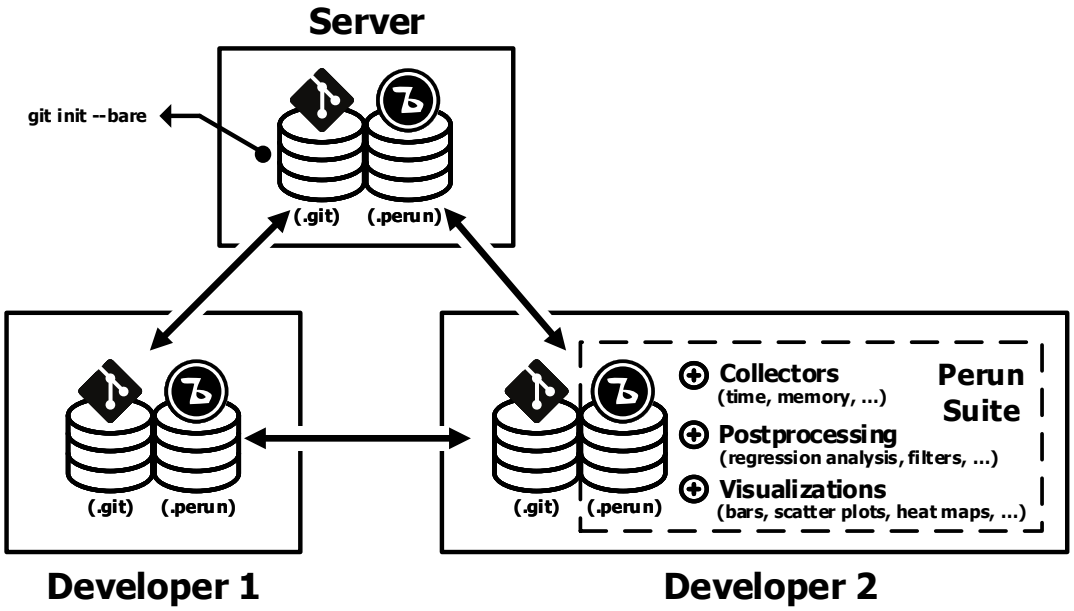
THIRTEEN

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.

13.1 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.

13.1.1 Version Control System API

`perun.vcs.`**`init`**(*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_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`**(*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 **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**()
Generator of major versions for the current wrapped repository.

This function is currently unused, but will be needed in the future.

Returns iterable stream of major version representation

perun.vcs.**get_minor_head**()
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*.

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**()
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.

Returns string representation of the major version

perun.vcs.**get_minor_version_info**(**args*, ***kwargs*)
Wrapper function of the @p func

perun.vcs.**check_minor_version_validity**(**args*, ***kwargs*)
Wrapper function of the @p func

perun.vcs.**massage_parameter**(*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

- **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

perun.vcs.**is_dirty**()
Tests whether the wrapped repository is dirty.

By dirty repository we mean a repository that has either a submitted changes to its index (i.e. we are in the middle of commit) or any unsubmitted changes to tracked files in the current working directory.

Note that this is crucial for performance testing, as any uncommited changes may skew the profiled data and hence the resulting profiles would not correctly represent the performance of minor versions.

Returns whether the given repository is dirty or not

perun.vcs.**save_state**()
Saves the state of the repository in case it is dirty.

When saving the state of the repository one should store all of the uncommited changes to the working directory and index. Any issues while this process happens should be handled by user itself, hence no workarounds and mending should take place in this function.

Returns

perun.vcs.**restore_state**(*saved*, *state*)
Restores the previous state of the the repository

When restoring the state of the repository one should pop the stored changes from the stash and reapply them on the current directory. This make sure, that after the performance testing, the project is in the previous state and developer can continue with his work.

Parameters

- **saved** (*bool*) – whether the stashed was something
- **state** (*str*) – the previous state of the repository

perun.vcs.**checkout**(*minor_version*)
Checks out the new working directory corresponding to the given minor version.

According to the supplied minor version, this command should remake the working directory so it corresponds to the state defined by the minor version.

Parameters **minor_version** (*str*) – minor version that will be checked out

13.1.2 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/e/phdwork/perun/git/docs/_static/templates/supported_module_names.py
+++ /mnt/e/phdwork/perun/git/docs/_static/templates/supported_module_names_collectors.py
@@ -6,7 +6,7 @@
         ))
     return {
         'vcs': ['git'],
-        'collect': ['trace', 'memory', 'time'],
+        'collect': ['trace', '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.

13.2 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
    |-- /logs
    |-- /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
```

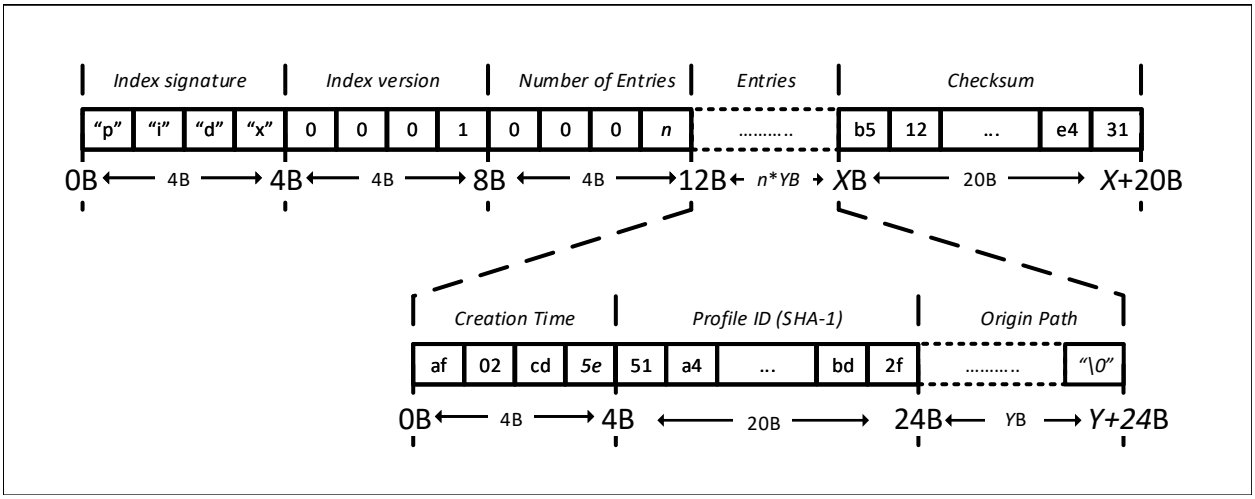
Each object from `.perun/objects` is represented by hash value, where the first two characters are used to specify directory and the rest of the hash value a file name, where the index or compressed file is stored.

`.perun/logs`: Contains various logs for various phases. Currently this holds logs for each minor version, for which we precollected new profiles during the `perun check` command. This behaviour can be set up by setting *`degradation.log_collect`* to true.

`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.

13.2.1 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.

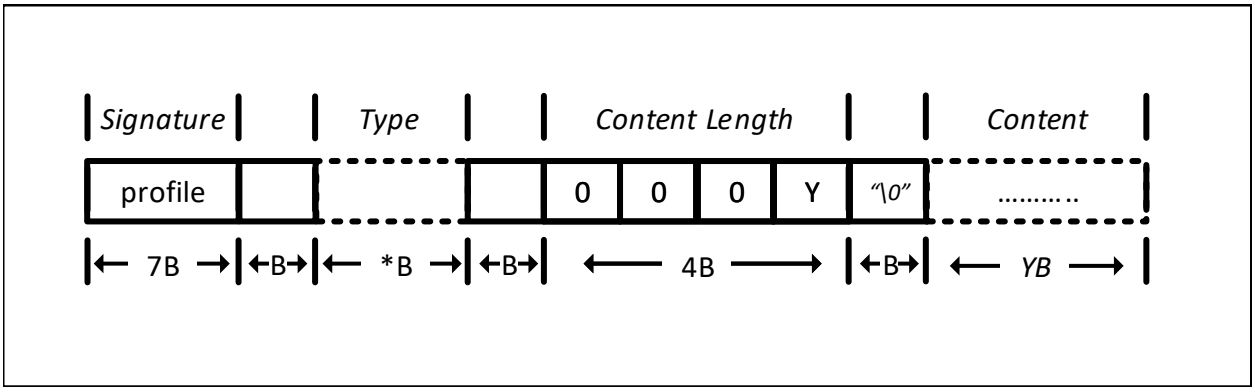
***Entries* [*variable length*]:** One entry of the index corresponds to one assigned profile. Each entry is of variable lenght and contains the identification of the original profile file, together with timestamp of creation and the identification of the compressed object, that contains the actual profiling 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.

13.2.2 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*.

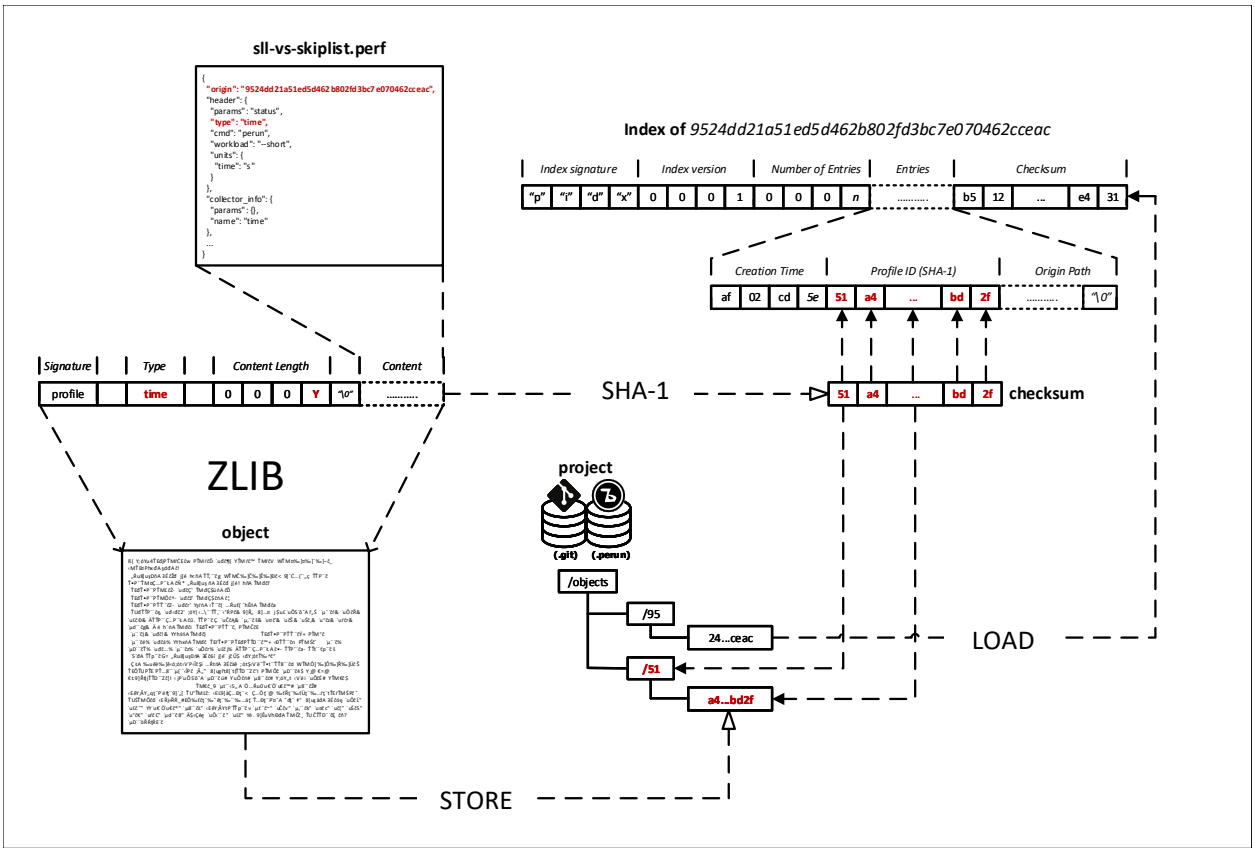
13.2.3 The Lifetime of profile: Internals

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

FOURTEEN

CHANGELOG

14.1 HEAD

To be included in next release

14.2 0.18.1 (2020-02-13)

commit: 5166ec167bddc5e72877029dc19f1302aaa8cb8b

Refactor trace collector

- refactor trace collector
- extend trace collector with watchdog module
- selected temporary files moved to .perun directory structure
- add diagnostic mode for trace collector
- add locking module to perun logic
- add diagnostic mode to tracer
- ignore tracer tests in codecoverage

14.3 0.18 (2020-02-11)

commit: 252078825559d67b6c915ab01827a9433af92dfa

Add performance fuzz-testing

- add `perun fuzz` mode implementing mutation based fuzzer. See *Performance Fuzz-testing* for more details.

14.4 0.17.4 (2020-01-28)

commit: cfb264ed53155a4aabf3874729c2c573533b0d1d

Add tabular view

- add tableof view module
- add conversion functions of models to dataframe

- add headers to tableof view
- add formats to tableof view
- add sorting to tableof view
- add filtering to tableof view
- add two modes of tableof (resources and models)
- fix minor bug in bounds collector (unknown collector type)
- fix templates for generating units

14.5 0.17.3 (2020-01-09)

commit: f3819834803b1eed878feeee51b697b260988c65

Add Loopus collector in Perun

- fix an issue in profiles which contained only persistent properties
- add bounds collector, wrapper over Loopus tool

14.6 0.17.2 (2019-08-16)

commit: d82d15ec5635dacf8027311e44aa65b4776dc8fb

Improve the runner logic

- extract cmd, args and workload to Executable class
- remove `--remove-all` argument in `perun rm`
- add support for removing profiles from pending jobs through perun
- improve the output of *perun rm* command
- extract CLI groups to isolate modules
- add caching to selected vcs commands
- fix untested bug in degradation check
- rename warmup parameter in *time* to `--warmup`
- lower the number of warmup and repetitions for time collector during tests
- remove filter postprocessor (did nothing)
- add signal handling to runner (authored by Jirka Pavela)

14.7 0.17.1 (2019-07-24)

commit: e2fad3cd2ac22f17aa7abfe4375d9940eb9f2847

Add new degradation detection methods

- add new detection methods for parametric and non-parametric models
- add **Integral Comparison** detection method, which computes the integrals under models

- add **Local Statistics** detection method, which analyses the various statistics in intervals of models
- refactor various minor issues in postprocessing logic
- add new strategies for detecting performance changes

14.8 0.17 (2019-07-09)

commit: e6b1e88d766d93cdab4f114464df51114d6415a8

Optimize profile format

- make profile format more compact
- fix minor issue in fast check
- extract selected functions from query to profile object

14.9 0.16.9-hotfix (2019-06-18)

commit: 126473caba3685878bf79f687115023918d5048a

Hotfix issue in Makefile

- hotfix issue in Makefile

14.10 0.16.9 (2019-06-18)

commit: 8e7228deb81cdfacfea7e7273fd25e70503cbe2b

Add CLI for stats manipulation

- refactor the perun stats module
- extend the stats module with a CLI
- add new operations (list, delete, . . .) to the stats module

14.11 0.16.8 (2019-05-18)

commit: 05d7275cb5f00183a72f8428bc4aaba0420bb73b

Extend perun instances with temporaries

- add new logic module that allows to store temporary files in separate directory (.perun/tmp)

14.12 0.16.7-hotfix (2019-04-15)

commit: 686ea87a64d845b215474193f879db0240c05732

Hotfix Jinja potential vulnerability

- hotfix Jinja potential vulnerability

14.13 0.16.7 (2019-04-15)

commit: 4152091bc4c2e5d1553ebccfef059d8153255aba

Extend perun instances with stats

- add new logic module that allows to store stats for profiles in separate directory (.perun/stats)

14.14 0.16.6 (2019-03-25)

commit: 18870d9d5853726d5cc966962d275111e451ab06

Improve the quality of life of Perun

- fix minor bug in storing changes
- extracted index entry specific functions to isolate class (in order to create new versions)
- implement index v2.0, codename FastSloth
- switch to working with index v2.0 (index v1.0 is still supported, however, everything is saved as 2.0)
- minor refactors
- optimize loading of the profile info for both registered and pending profiles (yields huge performance boost)
- add *–force* option to *perun add* which will force the add (d'oh)
- add printing of trace if *perun -vv* is set in cli (i.e. the verbosity is of level 2+)
- rename 'params' in profile to 'args' since it complies to other parts of code
- refactor minor issues, enhance error messages and exception handling

14.15 0.16.5 (2019-03-22)

commit: a2bd359479920178cfed1a0de779ef6fa5f4d4ac

Revive complexity collector

- revive the complexity collector
- increase the test coverage of complexity collector
- update the complexity collector to comply with latest version of Perun

commit: 983b02ca54faa66941dcea06f990c8033eaf98f6

Add kernel non-parametric regression

14.16 0.16.4 (2019-03-14)

commit: 983b02ca54faa66941dcea06f990c8033eaf98f6

Add kernel non-parametric regression

- fix minor issue in memory collector that manifests with gcc-5.5+ and Ubuntu 18.04+
- add three kernel non-parametrik regression models (see _postprocessors-kernel-regression)

- fix minor issues in moving average and regressogram

14.17 0.16.3 (2019-03-02)

commit: de699ab66d8438166d0ad2d55c74bf43a59d1fc4

Overhaul the trace collector

- update to Click version 7.0 (because underscores are replaced by dashes)
- add automatic pairing of the static probes in trace collector
- add fault-tolerant system to trace collector (now it does collect some profile even if it contains some corruption)
- rework the internal format of traces

14.18 0.16.2 (2019-03-02)

commit: 77bed5eb7654274687fc0fa7130f28a6ff282fba

Fix and refactor the memory collector

- fix minor issue in average amount threshold checker, when average is 0
- refactor memory collector
- add proper documentation to memory collector
- fix an ubuntu 18.04 issue, when dlsym() needed some bytes before libmalloc.so is properly loaded resulting into crash
- add proper locking to memory collector

14.19 0.16.1 (2019-03-01)

commit: 04cd6a2dc788f73d21c23554ff2ab8174af67dbd

Add moving average postprocessor

- add moving average postprocessor, other of the non-parametric analysis
- minor fixes in regressogram (refactor and documentation)
- add *perun fuzz* command which does a performance fuzzing
- remodel runner functions to generators

14.20 0.16 (2019-02-16)

commit: 04cd6a2dc788f73d21c23554ff2ab8174af67dbd

Add regressogram postprocessor

- add –version option to perun cli, so it shows version of perun (d'oh!)
- extend scatterplot to support step function rendering (for regressogram)

- add regressogram postprocessor, one of the non-parametric analysis

14.21 0.15.4 (2018-08-13)

commit: b1e2e3bdcca839efcf7a59ebb8fdbd2b8fc38888

Add cleanup procedures to Trace collector

- add cleanup procedures to trace collector (so it properly kills systemtap modules)
- fix setup.py versions
- make clusterizer less verbose
- fix wrong parameter name in trace collector

14.22 0.15.3-hotfix (2018-08-02)

commit: a9b46ed478258bbcd8292df0775a14e69b7db329

Hotfix unused workload parameter in trace collector

- hotfix missing workload parameter in trace collector

14.23 0.15.3 (2018-08-01)

commit: a9b46ed478258bbcd8292df0775a14e69b7db329

Extract trace configuration automatically

- rename complexity collector to **trace**
- fix minor issues with trace collector
- add basic support for parallel programs in trace collector
- add basic support for non-terminating programs (–timeout) in trace collector
- fix minor issues in incorrect piping (class with ||)
- add lookup of profiled functions in trace collector

14.24 0.15.2 (2018-07-20)

commit: a9b46ed478258bbcd8292df0775a14e69b7db329

Upgrade Trace collector architecture

- update the cli of the *Trace Collector* with new options
- add support for static and dynamic probing of the binaries (hence allow custom user probes)
- fix minor issues
- rework the architecture of system-tap collector to work as a daemon

14.25 0.15.1 (2018-07-17)

commit: f137abac6c428fc5e580dfa0fc9446c65ac30e4c

Rehaul the notion of workloads

- refactor check modules
- add `pending tag range` to `perun add` command to add more profiles at once
- add `index tag rage` to `perun rm` command to remove more profiles at once
- fix the issue with wrong sort order and tags (now *`format.sort_profiles_by`* sets the option in local)
- add support for workload generators
- implement integer workload generator that generates workload from the integer interval
- implement singleton workload generator that generates single workload
- implement string workload generator that generates random strings
- implement file workload generator that generates random text files
- add *`generators.workload`* for specification of workload generators in config
- remodel the notion of workloads to accept the workload generators to allow other style of workloads
- add two modes of workload generation (one that merges the profiles into one; and one which gradually generates profiles)
- add default workload generators to shared configuration

14.26 0.15 (2018-06-20)

commit: 6bb792fd8e172ab6c97a3cd1ac517bfe416b6c85

Extend the suite of change detection methods

- add fast check degradation check method (*Fast Check*)
- add linear regression based degradation check method (*Linear Regression*)
- add polynomial regression based degradation check method (*Polynomial Regression*)
- rename regression models to full names
- fix divisions by zero in several places in regression analysis
- rename the api of several regression functions

14.27 0.14.4 (2018-06-17)

commit: 4e36142252e123f3e8e6422583c71383adc9fc30

Refactor the code

- fix various linting issues (e.g. too long lines)
- remove unused code and function (e.g. in memory)
- fix minor issues

- extend the test suite with several more tests
- flatten the test hierarchy
- remove alloclist view (query+convert imported in python is more powerful)
- renew the rest of the old documentation format
- extract path and type function parameters from vcs api
- refactor pcs module and remove pcs as argument from all of the functions
- fix various codacy issues
- refactor cli module by moving callbacks, renaming functions and removing redundant functions

14.28 0.14.3 (2018-06-12)

commit: a2820c0cb50cff5b758a3d01ca7b8e356af5d2cf

Extend utils module

- print timing of various collection phases
- add `degradation.log_collect` to store the output of precollect phase in isolated logs
- add working `--compute-missing` parameter to check group, which temporarily sets the precollection
- add repetition of the time collector
- add predefined configuration templates
- add automatic lookup of candidate executable and workloads for user configuration (see *Predefined Configuration Templates*)
- add `perun config reset` command to allow resetting of configuration to different states
- extend the utils module with ELF helper functions
- extend the utils with non-blocking subprocess calls
- extend the utils with binary files lookup

14.29 0.14.2 (2018-05-15)

commit: 0faaa74097a159c4b441d65415dba504265c2059

Rehaul the command line output

- fix issue with pending tags not being sorted ;)
- fix the issue with incorrectly flattened values in query
- extend the memory collector to include the allocation order as resource
- add loading and storing of performance change records
- add short printed results for found degradations
- update the default generated config
- remake the output of time collector
- fix issue with integer workloads

- fix issue with non-sorted index profiles
- fix issue with memory collector not removing the unreachable allocations
- add vcs history tree to log (prints the context of the vcs tree)
- remodel the output of the degradation checks
- switch the colour of optimizations to green (instead of blue)
- colour tainted (containing degradation) and fixed (containing optimization) branches in vcs history
- add short summary of degradations to each minor version in graph
- add semantic ordering of uids (used in outputs)
- add vcs history to output of perun run matrix
- make perun check precollect phase silent (until we figure out the better way?)
- add streaming to the history (so it is not output when everything is done)
- make two versions of run_jobs (one with history and one without)
- refactor some modules to remove unnecessary dependencies
- add information about degradations to perun status and log

14.30 0.14.1 (2018-04-19)

commit: b7922d7c1bbe7ea89fe735c93cf1e6c8a7604765

Extend the automation

- add two new options to regression analysis module (see *Regression Analysis* for more details)
- fix minor issues in regression analysis and scatter plot module
- fix issue with non-deterministic ordering in flattening the values by convert
- add different ordering to perun status profiles (now they are ordered by time)
- add more boxes to the output of the perun status profiles (bundled per five profiles)
- add `format.sort_profiles_by` configuration key to allow sorting of profiles in `perun status` by different keys
- add `--sort-by` option to `perun status` to allow sorting of profiles in `perun status`
- fix minor things in documentation
- add few helper function for CLI and profiles
- rename origin in ProfileInfo to source (class of names)
- fix typos in documentation
- remake walk major version to return MajorVersion object, with head and major version name
- add helper function for loading the profile out of profile info
- extend the api of the vcs (with storing/restoring the state, checkout and dirty-testing)
- add `profiles.register_after_run` configuration key to automatically register profiles after collection
- add `execute.pre_run` config key for running commands before execution of matrix
- add helper function for safely getting config key

- add `--minor-version` parameter to `perun collect` and `perun run` to run the collection over different minor version
- add `--crawl-parents` parameter to allow `perun collect` and `perun run` to collect the data for both minor version and its predecessors
- add checking out of the minor version, and saving the state, to collection of profiles
- add *`degradation.collect_before_check`* configuration key for automatically collect profiles before running degradation check

14.31 0.14 (2018-03-27)

commit: 3e56911baad6a7cd0ab0b90b23c6edbc57abeb43

Add clusterization postprocessor

- add clusterizer postprocessor (see *Clusterizer*)
- add helper function for flattening single resources
- fixed profiles generated by time in tests

14.32 0.13 (2018-03-27)

commit: 9642c1dcd7ba39b91ef791039690f5be79312dd2

Add SystemTap based complexity collector

- add SystemTap based complexity collector (see *Trace Collector* for more details)
- 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

14.33 0.12.1 (2018-03-08)

commit: 96ef4443244568260e5dd25fa4cde5230eba8a36

Update project readme

- update the project readme
- add compiled documentation

14.34 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

14.35 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

14.36 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

14.37 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

14.38 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

14.39 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

14.40 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

14.41 0.9 (2017-08-31)

Add regression analysis postprocessing module

commit: 2b3d0d637699ae35b36672df3ce4c14fa0fed701

- add regression analysis postprocessor module
- add example resulting profiles

14.42 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

14.43 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

14.44 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

14.45 0.8 (2017-07-03)

Add flame graph visualization

commit: 56a29c807f2d7ad34b7af6002e5ebf90c717e8d7

- add flame graph visualization module

14.46 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

14.47 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

14.48 0.7 (2017-06-26)

Add bar graph visualization

commit: a0f1a4921ecf9ef8f5b7c14ba42442fc589581ed

- integrate bar graph visualization

14.49 0.6 (2017-06-26)

Add Flow graph visualization

commit: 5683141b2e622af871eabc1c7259654151177256

- integrate flow graph visualization

14.50 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

14.51 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

14.52 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

14.53 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

14.54 0.4 (2017-03-17)

Add Complexity collector

commit: 323228f95050e52041b47af899eaea6e90eb0605

- add complexity collector module

14.55 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

14.56 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

14.57 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

14.58 0.0 (2016-12-10)

Initial minimalistic repository

commit: 2a6d1e65e5f3871e091d395789b9fd44450ef9e4

- empty root

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

M

O

P

R

S

T

U

V

W