European Commission | Horizon 2020 European Union funding for Research & Innovation

GA no. 671657

D3.1
Final RRL architecture

Document type:	Report
Dissemination level:	Public
Work package:	WP3
Editor:	Andreas Gocht(TUD)
Contributing partners:	TUD,TUM
Reviewers:	
Version:	0.1

Document history

Version	Date	Author/Editor	Description
0.1	12-Jun-2017	Andreas Gocht(TUD), Anamika Chowdhury(TUM)	Inital version
0.2	18-Aug-2017	Andreas Gocht(TUD), Anamika Chowdhury(TUM), Per Gunnar Kjeldsberg(NTNU), Riha Lubomir(IT4I)	First review
1.0	30-Aug-2017	Andreas Gocht(TUD), Anamika Chowdhury(TUM), Per Gunnar Kjeldsberg(NTNU), Riha Lubomir(IT4I)	Final Version

Executive Summary

The objective of WP3 is to implement the READEX Run-time Library (RRL). Besides the RRL architecture implementation, this involves also the development of an efficient scenario detection and switching mechanisms and the development of an efficient run-time scenario calibration mechanism.

This deliverable describes the architecture of the RRL as well as the changes made to the two other tools we are using: Score-P and the Periscope Tuning Framework (PTF). Both tools interact with the RRL. The RRL itself is designed with modern programming standards in mind. The modular architecture of the RRL allows re-usage of different components during design time and runtime. This avoids code duplications and reduces the sources for errors. Furthermore, we show that the runtime overhead of the RRL is negligible.

For PTF we describe the changes made to support runtime situations during the tuning processes.

Finally, we outline the changes made to Score-P. These changes were made to allow a fast development of the RRL, which is supposed to be independent from the development of Score-P.

It is assumed that the reader of this document has already a good understanding of the READEX concepts from reading previous deliverables such as D4.1 and D4.2.

Contents

1 Introduction **4**

2 Architecture of the RRL **5**

2.1 Score-P 6
2.2 Control Center 6
2.3 OA Event Receiver 6
2.4 RTS Handler 7
2.5 Tuning Model Manager 8
2.6 Parameter Controller 9
2.7 Parameter Control Plugins 9
2.8 Calibration 9
2.9 Overhead 10

3 Communication with the Periscope Tuning Framework **11**

4 Improvements to Score-P **12**

4.1 Substrates Design Criteria 13
4.2 Calls to Plugins 14
4.3 Introduced Overhead 15

5 Summary **16**

1 Introduction

The objective of WP3 is to implement the READEX Run-time Library (RRL). Besides the RRL architecture implementation, this involves also the development of an efficient scenario detection and switching mechanisms and the development of an efficient run-time scenario calibration mechanism. However, this deliverable only describes the final RRL architecture. The other two points will be covered in Deliverable 3.2.

Figure 1 shows all components of the READEX architecture. In this deliverable, we will mainly target the architecture of the RRL itself, as well as the changes to the PTF and Score-P needed for the RRL. Some parts of this deliverable are already presented in the pre-alpha [?] report and Deliverable 4.2 [?]. Nonetheless, there have been significant changes to the different components.

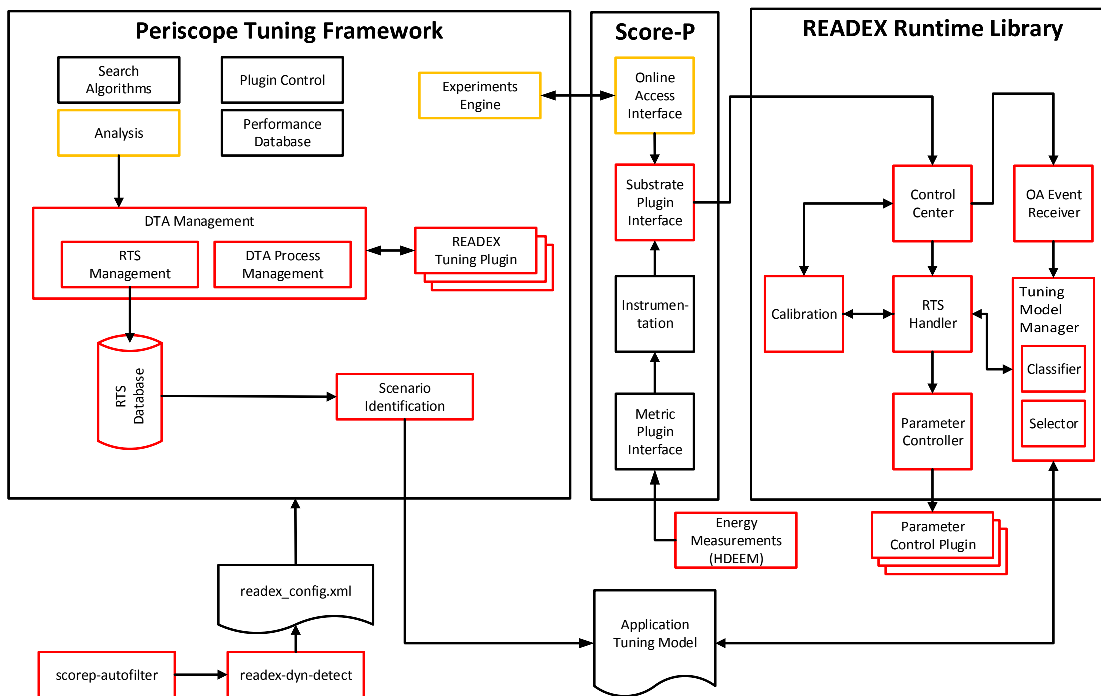

Figure 1: Overview of all components of the READEX tool suite

The deliverable is structured as follows: Section 2 will outline details about the implementation of the RRL. Section 3 presents details about the communication between PTF, Score-P and the RRL to reflect how the RRL is invoked during Design Time Analysis (DTA). In section 4 we will present the changes to Score-P needed to implement the RRL.

2 Architecture of the RRL

The design goal of the READEX Runtime Library (RRL) was to create an easy to modify and easy to extend library. To achieve this, different modern programming idioms like resource acquisition is initialization (RAII) have been used. Moreover, we choose to rely on standardized programming languages and communication formats. For the implementation of the RRL we choose the c++14[?] standard, which is supported by all state of the art compilers. For communication between PTF and the RRL, as well as for the Application Tuning Model we rely on the JSON standard[?][?]. To ensure maintainability of the RRL, the code is well documented using the Doxygen format[?].

Figure 2 shows the different modules that are implemented in the RRL. Each module has well defined interfaces.

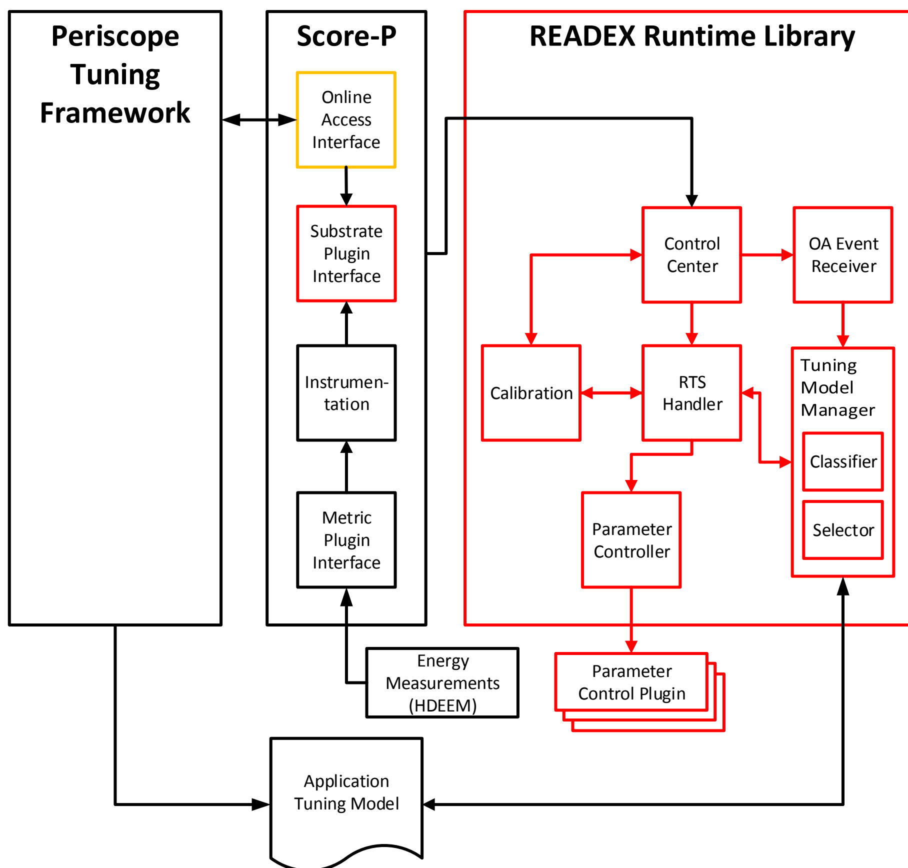

Figure 2: Overview of the RRL. All components in red are implemented as part of the READEX project

During the Design Time Analysis (DTA) and the Runtime Application Tuning (RAT) different parts of the RRL are used. The OA Event Receiver, which can be seen in Figure 2 is just present during DTA, whereas the Calibration is just present during the RAT. Furthermore, there is a different implementation for the Tuning Model Manager (TMM) for DTA and RAT.

Other parts of the RRL are shared between DTA and RAT. This is because different steps like instrumentation, Runtime Situation detection (rts detection) and setting of configurations are similar for both DTA and RAT. Therefore, we choose this modularised approach. It allows us to avoid code duplication of core components like the RTS Handler, which cares about rts detection, or the Parameter Controller, which sets the configurations.

Finally, this leads to an architecture, which is easy to debug and which allows fast changes if required. The following sections will describe the different components in more detail. Where needed a separation between DTA and RAT will be made. It is assumed that the reader of this document has already a good understanding of the READEX concepts from reading previous deliverables such as D4.1 and D4.2.

2.1 Score-P

Score-P is one of the key tools we are relying on on in READEX. It is able to instrument different programming languages like C, C++ and Fortran. The RRL is implemented as a Score-P Substrate Plugin. Please read Section 4 for details about the Substrate Interface.

The interface is written in C. The RRL builds an abstraction around this C interface. This abstraction takes care of the instantiation of the Control Center and the forwarding of Score-P events like *Enter Region* and *Exit Region*. Moreover, it provides functions to access different Score-P functions, which handle for example the translation between Score-P internal region id's and human readable region names.

2.2 Control Center

The Control Center acts as basic interface between Score-P and the rest of the RRL. Besides the instantiation of the different subcomponents like the Tuning Model Manager (TMM) or the RTS Handler, it decides which module gets a particular information. The different enter, exit and user parameter events are forwarded to the Calibration module and the RTS Handler, whereas the so called *Generic Commands* are redirected to the OA Event Receiver.

2.3 OA Event Receiver

The OA event receiver provides the interface between the PTF and the RRL and it is used only during the DTA. PTF generates tuning requests, which are propagated to Score-P through the so called Online Access interface (OA interface). Please see Section 3 for details regarding the communication between PTF and Score-P.

When a tuning request is received by Score-P, the OA interface generates an event and forwards the tuning request as a *Generic Command*. The information forwarded is a JSON encoded string, which is then parsed by the OA Event Receiver.

In the next step, the OA Event Receiver parses and forwards this information to the TMM. Listing 1 shows the current format of an OA Tuning Request. Besides different identifiers like the *Event Type* the request contains information about the configuration requested for different rts's. The configuration is then passed to the Tuning Model Manager.

Listing 1: Format of the OA tuning request (version 0.2)

```
{
    ”genericEventType”:”tuningRequest”,
    ”genericEventTypeVersion”:0.2,
    ”data”:
    {
        ”RegionCallpath”:
        [
            {
                ”RegionID”:<Region Id>,
                ” AdditionalIdentifiersInt ”:
                {
                    <Name1>:<Value1>, [...]
                },
                ”AdditionalIdentifiersUInt”:
                {
                    <Name1>:<Value1>, [...]
                },
                ” AdditionalIdentifiersString ”:
                {
                    <Name1>:<Value1>, [...]
                }
            },
            [...]
        ],
        ”TuningParameters”:
        {
            ”coreFrequency”:<Frequency setting>, [...]
        }
    }
}
```

2.4 RTS Handler

The RTS Handler maintains the current call stack and collects the Additional Identifiers, like matrix sizes or hints that lead to a different function behaviour. Upon receiving an

enter region notification from Score-P during the application run, the RTS Handler checks whether the current region is a significant or an unknown region. If it is an unknown region, calibration is kicked off, and nothing else is done.

If the region is significant, the RTS Handler shall receive the next configuration to use from the TMM. For this a complete rts is needed consisting of a call path together with Additional Identifiers. The RTS Handler sends the current call path to the TMM, which returns the number of additional identifiers that are expected. Once these are collected, the RTS Handler request a new configuration from the Tuning Model Manager using the generated rts. This configuration is then passed to the Parameter Controller (PC).

Upon receiving an exit region event, the RTS Hander checks if the current region was set up for calibration. If yes, it requests the configuration for the currently exited region from the Calibration Module and saves this information to the TMM. If the region was not set up for calibration, the PC is informed that it might want to unset the current configuration. However, the actual decision whether the configuration gets unset or not is left to the PC.

2.5 Tuning Model Manager

The Tuning Model Manager has two different implementations. While the first implementation is used during DTA, the second one is used during RAT. It is possible, but not planned, to implement additional Tuning Model Managers.

2.5.1 DTA Tuning Model Manager

The DTA Tuning Model Manager is a lightweight version, which only saves configurations for individual rts's. It does not do any classification or scenario selection. This allows the RRL to provide exactly the configuration that is requested by PTF.

Because this Tuning Model Manager is intended to be used with PTF, it does not load or save any configuration files. During DTA it is filled with configurations from the OA Event Receiver.

2.5.2 RAT Tuning Model Manager

The RAT Tuning Model Manager (RAT TMM) implements scenario classification and selection. During initialization of the RAT TMM the Tuning Model (TM) is loaded.

The RAT TMM checks if the rts received from the RTS Handler is present in the Tuning Model. If yes it gets the scenario associated with the given rts. To each scenario a certain configuration is associated. This configuration is then handed back to the RTS Handler. Details regarding more advanced scenario classification and configuration selection will be given in Deliverable 3.2.

Currently, clustering of the different rts's into scenarios is only done during DTA by PTF and transferred to RAT through the TM. The RAT TMM in the RRL then selects these scenarios based on the rts's.

However, together with the calibration mechanism the RAT TMM will be able to add new configurations to the TM. Further details will be given in Deliverable 3.2.

2.6 Parameter Controller

The Parameter Controller takes care of the loading, setting and finalizing the Parameter Control Plugins (PCP). It gets a configuration to apply from the RTS Handler. The Parameter Controller supports two different modes: *reset* and *no-reset*. The first mode maintains a configuration stack. Whenever a new configuration is set, this configuration is pushed onto a stack. When the corresponding unset occurs, the element is removed from the stack and the previous configuration is set.

If the *no-reset* mode is set, a new configuration stays active until a new configuration is set. The unset is ignored. The behavior is configurable using an environment variable and managed by a so called *configuration manager*, which has two different implementations. It is possible, but not planned, to implement additional configuration managers.

2.7 Parameter Control Plugins

The Parameter Control Plugins (PCP's) perform the configuration of different hardware resources.

Each plugin is loaded, initialized, and used by the by the Parameter Controller. The RRL defines an Interface, which allows the users to build their own PCP's.

Currently there are the following plugins available:

- Dynamic Voltage and Frequency Scaling (DVFS)
- Uncore Frequency Scaling (UFS)
- Energy Performance Bias (EPB)
- MPI
- OpenMP

A detailed description of the different Tuning Plugins can be found in Deliverable 1.1 [?].

2.8 Calibration

The goal of the calibration is to handle unseen runtime situations (rts) during the production runs of a HPC application. These unseen rts's may consists of already known regions, which

have some unknown parameters. Alternatively, these rts's may consist of completely new regions, which have totally unknown characteristics.

The calibration module and the related mechanisms will be part of Deliverable 3.2.

2.9 Overhead

To determine the overhead caused by the RRL we measured the runtime of Score-P events. We choose events generated by the Score-P user instrumentation. Each user instrumentation call generates a Score-P event, which is propagated to the RRL. The benefit of these user events is that they give an easy measurement of the runtime and a separate measurement of enter and exit calls. We introduced two different user regions called "foo" and "bar" as well as the necessary OA phase for the OA interface. Listing 2 shows the measurement of an Score-P enter event.

Listing 2: Measurment of an enter region call

```
1 start = std::chrono::high_resolution_clock::now();
2 SCOREP_USER_REGION_BEGIN(foo, "foo", SCOREP_USER_REGION_TYPE_COMMON);
3 elapsed_ns = std::chrono::high_resolution_clock::now() - start;
4 timing.foo_start = elapsed_ns.count();
```

Moreover, we provided a Tuning Model and Parameter Control Plugins for the core frequency and the uncore frequency. Finally we choose the *no-reset* behaviour of the Parameter Controller. This allowed us to measure the full overhead generated by the RRL during runtime together with the Parameter Control Plugins.

Region Event	Runtime in μs
OA phase enter	0.152
OA phase exit	0.148
foo enter	0.130
foo exit	0.121
bar enter	0.130
bar exit	0.121

Table 1: Runtime of Score-P user instrumentation

Region Event	Runtime in μs
OA phase enter	8.588
OA phase exit	2.281
foo enter	8.611
foo exit	2.338
bar enter	8.563
bar exit	2.310

Table 2: Runtime of Score-P user instrumentation, RRL and Tuning Model

Table 1 shows the overhead generated by Score-P itself, which is quite low. Table 2 shows the overhead of the RRL together with the Parameter Control Plugins. It can be seen that the runtime varies depending on whether there is an enter event or exit event of a region. This is caused by the Parameter Control Plugins, which are only invoked during the entering of the region, as we choose the *no-reset* behaviour.

However, in both cases, enter and exit, the runtime is quite small compared with the runtime of a region we consider as significant. Significant regions are at least $100ms$ long. A combined overhead of around $11\mu s$, which is 0.011% of the runtime of a significant region, is still negligible.

3 Communication with the Periscope Tuning Framework

The Periscope Tuning Framework (PTF) is a distributed framework that enables to perform performance analysis and tuning with respect to various tuning aspects.

PTF consists of a frontend and a hierarchy of analysis agents. The frontend actuates the performance analysis by loading the tuning plugins. These plugins request different measurements and different settings from the analysis agents to tune the user application for different objectives. The analysis agents are connected to the application through the OA interface of Score-P [?]. These agents collect performance data from the application processes and apply different settings to the application.

During DTA, PTF determines the best configurations by applying different analysis strategies that are executed by the analysis agents. To do so, PTF sends two types of requests to Score-P through the OA interface:

- measurement requests,
- tuning request.

The READEX tuning plugin sends the measurement request to the analysis agents for several global metrics, such as execution time, PAPI counters or rusage. Additional metrics accessible by the Score-P metric plugins (e.g energy measurements) are also supported. The agents

then forward the request to the OA interface. Listing 3 shows an example for a measurement request sent from PTF to Score-P through the OA interface. The request triggers Score-P to collect node energy from the HDEEM energy plugin[?]:

Listing 3: Measurement request sent from PTF to Score-P

```
1 <REQUEST GLOBAL METRIC OTHER "hdeem/BLADE/E">
```

The OA interface of Score-P parses the request and registers the metric request with the measurement system of Score-P. The Score-P profiling system collects the measurement results of the metric for all the regions of the application and sends them back to PTF through the OA interface.

During DTA, PTF also configures the tuning parameters. It measures the effect of the new configuration on the tuning objective. The objective is evaluated for each rts of a significant region. Similar to the measurement request, a tuning request is sent by PTF to Score-P. The OA interface in Score-P converts the tuning request to a generic event, which is picked up by the RRL, which applies the new configuration.

While previous version of the OA interface where able to handle regions only, the new version is able to handle rts's as well. In contrast to simple regions, an rts encode the whole call path together with additional identifiers provided by the user. The different elements of a call path are encoded as Score-P region ids. Additional identifiers are encoded as strings. The type of these additional identifiers is limited to integer, unsigned integer and string.

Listing 4 shows an rts tuning request sent from PTF. The request encodes a switch from the current CPU core frequency to 1.2 GHz. The rts call stack consist of one element with the Score-P region id 12. Moreover, it is valid only if the additional identifier *A* of type *int* has a value of 14.

Listing 4: Tuning request from PTF to the Score-P OA interface

```
1 <RTSTUNINGREQUESTS((12,INTPARAMS=("A"=14),UINTPARAMS=(),STRINGPARAMS=()))=\\("CPU\_FREQ"=1200000)>
```

This tuning request is later converted to a JSON formatted string and emitted as a generic event. Listing 5 shows the JSON string generated from the request shown in Listing 4.

4 Improvements to Score-P

To allow the implementation of the RRL, an interface is needed to expose the functionality of Score-P. Score-P already uses an internal so called substrate interface. However, implementing an internal substrate requires recompilation of the measurement environment and an integration in the Score-P source code tree. This is impractical for a fast development of the

Listing 5: Tuning request from the Score-P OA interface to the RRL


```
{
    "genericEventType":"tuningRequest",
    "genericEventTypeVersion":0.2,
    "data":
    {
        "RegionCallpath":
        [
            {
                "RegionID":12,
                " AdditionalIdentifiersInt ":
                {
                    "A":14
                },
                "AdditionalIdentifiersUInt":
                {},
                " AdditionalIdentifiersString ":
                {}
            }
        ],
        "TuningParameters":
        {
            "CPU_FREQ":1200000
        }
    }
}
```


RRL using the design criteria described in Section 2. Thus, we provide a plugin interface to dynamically access the internal substrate functionality. Moreover, this plugin interface can be used to implement different other online plugins, which for example can help to understand and visualise the application behaviour. In this section, we describe the interface itself.

4.1 Substrates Design Criteria

Different substrates put diverging demands on the information that is provided by the monitoring infrastructure. Thus, Score-P must not only pass the incoming events to the registered plugins, but must also provide information about the supplied data. With the proposed interface, substrate plugins can register for specific types of events. These cover general events like the entering and exiting of a function, but also specialized events that are related to specific adapters. With each of these events, plugins receive a minimal set of information, which is an identifier for the thread whose monitoring issued the event and the timestamp associated with it. Further data depends on the type of the event that is monitored and can for example include information about the communication partner (e.g., for MPI events) or a set of strictly synchronous metrics (e.g., for enter and exit events). Substrate plugins may

choose to register only for those events that are relevant to them. Additionally, they can query the Score-P runtime for meta-data about the supplied information, e.g. the type and name of the thread where the current event occurred.

If the monitoring is distributed among different processes, plugins should also be able to communicate to enable a global view of the current state. Score-P enables plugins to use an internal interface for multi processing paradigm (MPP) communication. With this interface, processes can synchronize their state independent of the MPP used in the analyzed program. Currently this is not needed for the RRL. However, it might happen, that some kind of synchronization is needed, where this can be used.

Substrate plugins receive an event when the monitored application finishes, allowing them to write out the collected information. Likewise, when the monitoring is initialized, an appropriate call enables them to read existing configuration variables.

4.2 Calls to Plugins

We designed the interface in a way that enables programmers to access all relevant data to get a most comprehensive status for their monitoring or tuning implementations. The interface currently consists of three major parts:

1. The plugin definition, which provides callbacks to the substrate plugin for 15 management events,
2. A list of 62 application events that a substrate plugin can register for, and
3. A list of 46 callbacks to Score-P internals, that enable plugins to interpret events and synchronize the distributed state.

To register one or multiple substrate plugins, users set the environment variable `SCOREP_SUBSTRATE_PLUGINS`. When monitoring is initialized, Score-P reads this variable and attempts to load the respective libraries, in our case the RRL. If for example, the plugin `rrl` is registered, Score-P loads the shared object `libscorep_substrate_rrl.so`. Afterwards, it retrieves the plugin definition. Management events that are supplied with the plugin definition are stored for future reference. Afterwards, Score-P initializes the substrate by calling its `initialize` function. If the initialization fails, a warning is prompted and Score-P deregisters the plugin. If the initialization succeeds, plugins are supplied with callbacks to internal functions (`set_callbacks`). These can be used to retrieve internal information (e.g., the scope of a metric or the name of a location) and to access internal functionality like a synchronization mechanism, which transparently maps the calls to the used MPP. The usage of MPP functions should be delayed until the MPP is available, i.e., `initialize_mpp` is called. After Score-P callbacks are provided to the plugin, a list of functions for application events is gathered via the function `get_event_functions`. From this moment on, internal definitions (e.g., metrics or code regions) can be defined. Substrates receive such information via the `new_definition_handle` function. Later in the initialization phase, an identifier is

assigned to each substrate plugin via a call to `assign_id`. This identifier can later be used to store and retrieve thread-local data. Afterwards, the measurement is started and the plugin is able to retrieve the same management and application events as the existing substrates, profiling and tracing. When the monitoring ends, substrate can receive calls when Score-P is about to unify the collected monitoring data (`pre_unify`), when it flushes data to the file system (`write_data`) and when the measurement system is shut down (`finalize`).

An overview of the order of the management calls is depicted in Figure 3.

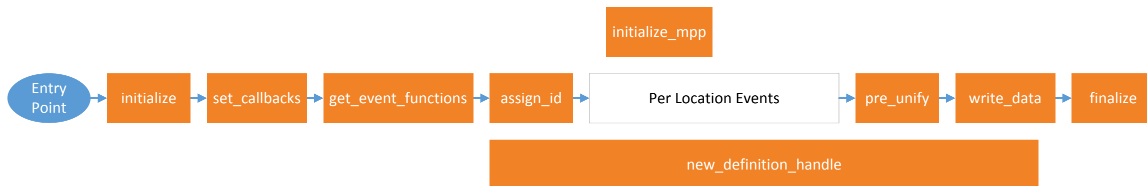

Figure 3: Order of calls to substrate plugin management functions. All functions except for the plugin definition (entry point) are optional.

5 Summary

The previous sections explained the architecture of the Readex Runtime Library. While the implementation of the different components might change, the architecture itself is deemed final. However, it might happen that changing requirements during the remaining project lifetime require changes to the RRL. We took care of this by using modern programming idioms and by introducing a flexible architecture.

Moreover, we improved the communication with the Periscope Tuning Framework. It is now possible to request a configuration for complex RTS's. The combined architecture for DTA and RAT reduces error sources and improves the consistency of results between DTA and RAT.

Finally, we reduced the dependency on Score-P releases by introducing the Substrate Plugin interface. Once this interface is released together with an upcoming Score-P release, the READEX approach can be adopted by all users that use Score-P. Moreover, changes to the RRL can be done and published quickly and independently from Score-P. This allows us to provide faster uptake of user suggestions.