GA no. 671657

D4.2
Prototype READEX Tool Suite

Document type:	Report
Dissemination level:	Public
Work package:	WP4
Editor:	Venkatesh Kannan (ICHEC-NUIG)
Contributing partners:	TUM, NTNU, TUD, ICHEC-NUIG
Reviewers:	Kai Diethelm (GNS)
	Zakaria Bendifallah (Intel)
Version:	0.3

Document history

Version	Date	Author/Editor	Description
0.2	27-Jan-2017	Andreas Gocht, Umbreen Sabir Mian (TUD), Michael Lysaght, Venkatesh Kannan (ICHEC-NUIG), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Mohammed Sourouri, Nico Reissmann (NTNU)	1^{st} review
0.3	13-Feb-2017	Andreas Gocht, Umbreen Sabir Mian (TUD), Michael Lysaght, Venkatesh Kannan (ICHEC-NUIG), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Mohammed Sourouri, Nico Reissmann (NTNU)	2^{nd} review

Contents

1 Introduction

The READEX project aims at developing a tools-aided approach for analysing and tuning HPC applications for energy efficiency on Exascale systems. To achieve this, the READEX tool-suite architecture and workflow were presented in Deliverable D4.1 [**?**]. In M14, the project delivered the pre-alpha prototype of the READEX tool-suite that demonstrated early prototypes of all components to verify their design and interfaces. It was limited to tuning significant regions for a single tuning parameter. In M18, the alpha prototype of the tool-suite was delivered which significantly extends the pre-alpha version with support for runtime situations (*rts*) and multiple tuning parameters. Optimized tuning configurations are determined for intra-phase dynamism. The extension for inter-phase dynamism is scheduled for the beta prototype in M30.

The alpha prototype of the READEX tool-suite implements the following features:

1. **Design-Time Analysis (DTA)**

   Here, the objective is to, first, analyse the tuning potential of a given HPC application. If the application exhibits significant dynamism that is quantified by dynamism metrics (execution time and compute intensity for the pre-alpha prototype), then it is considered to have significant tuning potential. Following this, the runtime situations of significant regions of an application with tuning potential are analysed for intra-phase dynamism using multiple objectives. The results of the DTA are used to create a *tuning model*.

   The tuning model encapsulates the results of the DTA in the form of *scenarios*, *configurations*, *classifier* and *selector*. While the *classifier* maps each instance of a significant region during its execution onto a *scenario*, the *selector* provides the best configuration determined for the tuning parameters during DTA for a given *scenario*.

   In the alpha prototype, the tuning experiments during DTA are performed for three tuning parameters, i.e., processor core frequency using DVFS (dynamic voltage and frequency scaling), processor uncore frequency using UFS (uncore frequency scaling) and number of OpenMP threads. The major extension for the DTA is to support runtime situations of significant regions while the pre-alpha prototype worked on region level only. It was agnostic for different instances of significant regions, e.g., reached via different call paths.

   The DTA is performed by the Periscope Tuning Framework (PTF) in conjunction with Score-P that provides instrumentation and measurement infrastructure. To perform the tuning actions when searching for the best configurations during experiments in DTA, PTF uses some modules implemented in the READEX Runtime Library (RRL), which is a new component created for Runtime Application Tuning (RAT) in READEX.

2. **Runtime Application Tuning (RAT)**

   Following the completion of DTA and creation of a tuning model, the READEX tool-suite can perform runtime tuning during the production run of the application. The runtime tuning is performed by first classifying each instance of a significant region into

a *scenario* as specified in the tuning model. In the alpha prototype, the scenario identification mechanism has been extended to use additional significant region identifiers in the form of Score-P user parameters that are included in the tuning model. Following this, the best configuration for the current scenario is selected and applied in a step called *switching*.

In the alpha prototype, the switching is performed for the processor core frequency, uncore frequency and number of OpenMP threads. The parameter control plugins that have been implemented to support this also additionally support energy performance bias (EPB) and number of MPI processes.

The RAT is performed by the RRL in conjunction with the Score-P tool.

The architecture of the READEX tool-suite is illustrated in Figure 1.

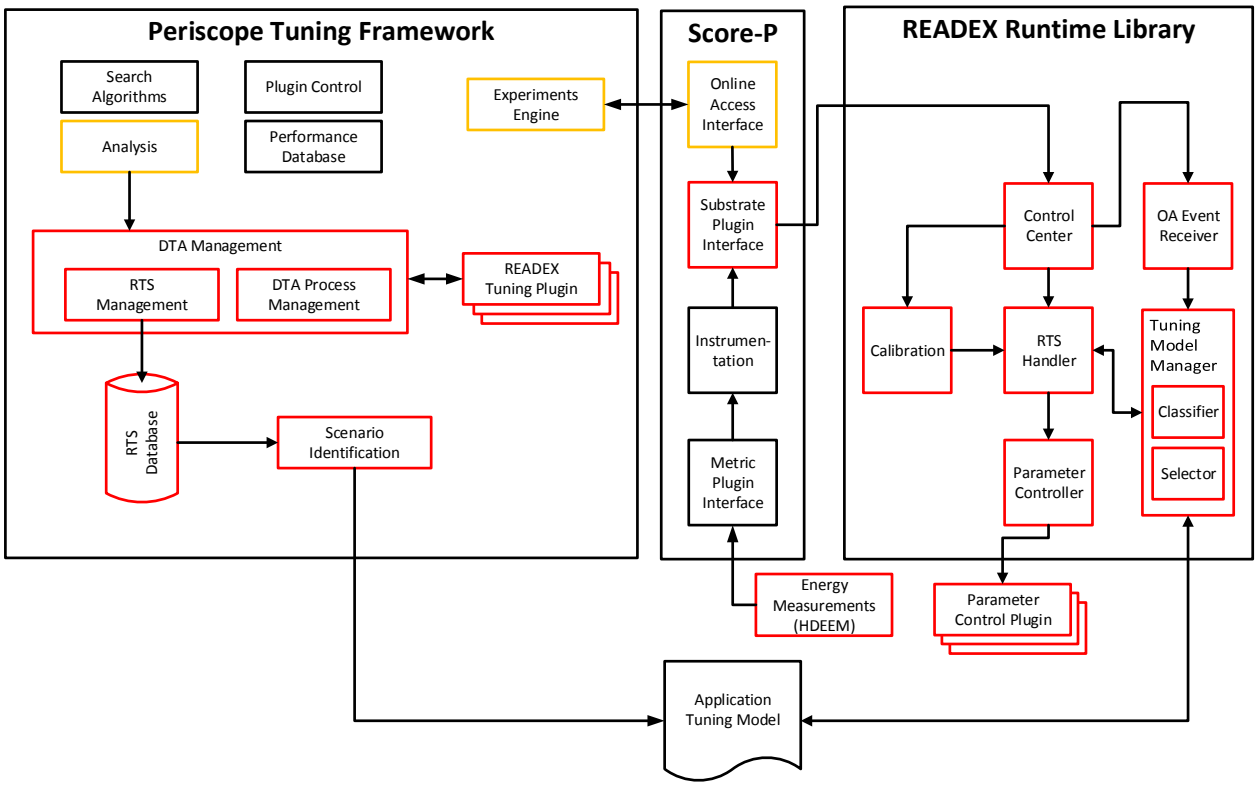

Figure 1: READEX tool-suite architecture

2 Features of Alpha Prototype

DTA identifies the best configurations of tuning parameters for runtime situations (*rts*'s), i.e., individual executions of program regions (see Deliverable D4.1 [**?**], page 10). These best configurations are stored in a tuning model which is then passed to the RAT which automatically switches to the identified configurations dynamically.

2.1 Features of Design-Time Analysis (DTA)

DTA forms the first stage in the READEX tool-suite. This section describes the new features that were added to the DTA stage. In the alpha prototype the following goals were achieved:

- Easy configuration of all DTA components via a single configuration file.
- Extension for *rts*'s. The pre-alpha prototype did tuning only on regions and had no support to determine system configurations for individual runtime situations.
- Extension for multiple tuning parameters. The pre-alpha prototype handled the optimisation of only the core frequency.

The internal components of Periscope and Score-P can be found in Figure 1. We now give a short overview of the functionalities added for each of the DTA components.

2.1.1 Scorep-autofilter

Scorep-autofilter automatically generates a filter file for runtime filtering of application regions to reduce measurement overhead. The work for scorep-autofilter focused on bug fixing.

2.1.2 Readex-dyn-detect

Readex-dyn-detect analyses an application for its dynamism. It is based on Score-P's tuple format which provides statistical information for runtime measurements. The result of readex-dyn-detect is a list of significant program regions with their available dynamism as well as a quantification of the available inter-phase dynamism.

The tool was extended with support for a common configuration file. The thresholds applied during the analysis are read from the READEX configuration file, and the significant regions found and the dynamism detected are added to the configuration file. The configuration file is then passed to DTA performed by PTF.

As part of the application work package, the RADAR report was developed to summarise the dynamism that is detected and the manual tuning results for the applications. Readex-dyn-detect was extended with the functionality to output the analysis results in LaTeX for their integration in the RADAR report.

2.1.3 Periscope

- DTA Management

  The DTA Management controls the overall execution of DTA. It receives *rts*'s from Score-P via its online access Interface and propagates them to the RTS database. It was significantly extended to support additional identifiers for *rts*'s. In the pre-alpha prototype the only identifier of significant regions was the callpath. The new version supports Score-P user parameters as additional identifiers. The application expert can distinguish instances of significant regions by defining user parameters for the region and assigning different values clustering different instances with similar runtime behavior.

  The DTA Management also reads the objective functions to be used for tuning from the READEX configuration file. The support for the following objective functions was added to DTA Management: Energy, Time, Core Energy, Energy Delay Product and Energy Delay Squared Product.

  The alpha prototype version then triggers the new READEX tuning plugin described below. When the intra-phase analysis is finished, the results are added to the RTS Database which is then used as input to generate the tuning model.

- RTS Database

  The RTS Database was modified to cover the extended *rts*'s with additional region identifiers. It now stores in addition to the best system configurations also information about the outcome for other tested system configurations to enable the tuning model generation to take more informed decisions.

- Experiment Engine

  It is responsible for carrying out the experiments to assess different system configurations. As DTA was extended in the alpha prototype to support tuning of *rts*'s, the experiment engine had to be extended as well for *rts* based system configurations. It submits *rts* based tuning actions via Score-P's online access interface to the RRL tuning substrate. In addition, PTF's analysis was extended for *rts* specific energy and performance measurements. These are returned to the READEX tuning plugin as a result of an experiment with a system configuration.

- Search Algorithms

  The search algorithms provided in PTF were extended to generate *rts* specific system configurations. The support was added to exhaustive, individual, random, and genetic search.

- Performance Database

  The Performance Database had to be extended for *rts* specific performance data. An experiment returns energy and performance data for *rts*'s which are stored in the database and used in the evaluation of the extended energy consumption property which is used in the READEX tuning plugin to determine the value of the objective functions.

PTF's Performance Database now also supports generic energy counters for core and node energy. This enables easier porting to other systems since the Taurus metric plugin of Score-P will have to be replaced by other system specific metric plugins, e.g., based on the RAPL counters. These plugins will support different metrics and an explicit support in PTF for all those metrics is not very efficient.

- READEX tuning plugin

  The alpha prototype comes with a new tuning plugin for READEX. It supports multiple objective functions, multiple search algorithms, multiple tuning parameters, and *rts*'s.

  The objective functions are forwarded by the DTA management to the tuning plugin. The plugin then initializes the search algorithm selected in the READEX configuration file and generates different *rts* based system configurations. These are forwarded to the Experiment Engine and the measured performance and energy data are returned to the plugin. The plugin finally computes the best configuration and adds all the results for all configurations to the RTS Database. The plugin tunes for the first objective function, but measures also the additional functions.

  The READEX tuning plugin supports the following tuning parameters: core frequency, uncore frequency, and OpenMP threads. The configuration of those tuning parameters is given in the common READEX configuration file, e.g., the frequency range is specified for the core and uncore frequency tuning parameters.

- Scenario Identification

  As a final step of the DTA the tuning model is generated from the RTS Database. The extensions in the alpha prototype are described in Section 2.2.

2.1.4 Score-P

The Online Access Interface was extended for *rts* specific tuning action requests based on a new *rts* based generic event in Score-P that is used to communicate these tuning actions to the RRL.

2.2 Features of the Tuning Model

The purpose of the tuning model is to connect DTA and RAT. In the pre-alpha prototype this was achieved by developing two modules: *scenario identification* and *tuning model*. The scenario identification module clustered *rts*'s with the same system configuration into *scenarios*, and the tuning model was used to save the scenarios and system configurations to disk as a file. In order to make the tuning model human-readable it is saved in the JSON data format. Additionally, the *Tuning Model Manager* (TMM), which is part of the READEX Runtime Library (RRL), was developed as part of the pre-alpha prototype. It deserialises the tuning model during RAT as explained in [3].

In the pre-alpha prototype tuning model, *rts*'s were differentiated only by their region name.

Comment[VK]: According to DTA Management in Section 2.1.3, identifier in pre-alpha is callpath, while in alpha it is extended to include Score-P user parameters. Please verify.

Comment[PGK]: Answer: In the pre-alpha version PTF took callpath into account. Because it was not used in the RRL, it was not included in the tuning model. Hopefully the current reformulation is less confusing.

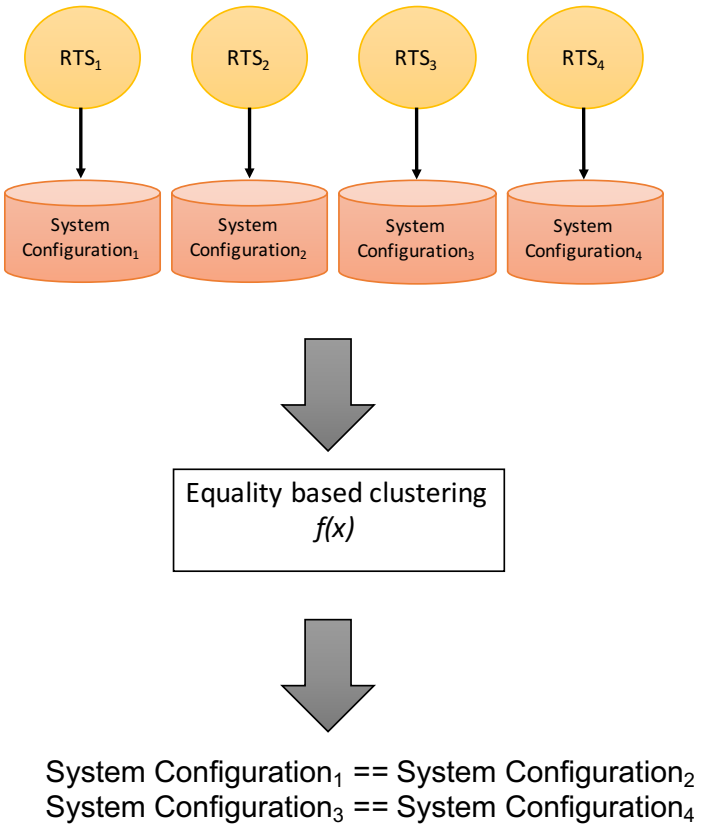

Figure 2: A high-level overview of the scenario identifcation process where *rts*'s and system configurations are created.

In the alpha version this is extended to also include the callpath to the region and Score-P user parameters. We describe the current version of the scenario identification and the tuning model in Sections 2.2.1 and 2.2.2, respectively.

2.2.1 Scenario Identification

The RTS Database generated by PTF contains all *rts*'s and their best found configurations (*cfg*). Each *cfg* holds information about the processor core frequency, uncore frequency and number of OpenMP threads. To reduce the size of the generated tuning model file, we cluster identical *cfg*'s. In both the pre-alpha and alpha prototypes this is done by checking for equality. For the beta prototype at M30, clustering based on similarity will also be included.

Now that the redundant configurations are evicted, we are ready to create scenarios for the different configurations.

Comment[KD]: In Figure 2, what is the meaning of the two green arrows, and why are they placed in this particular way?

Comment[PGK]: Answer: Confusing arrows are now removed.

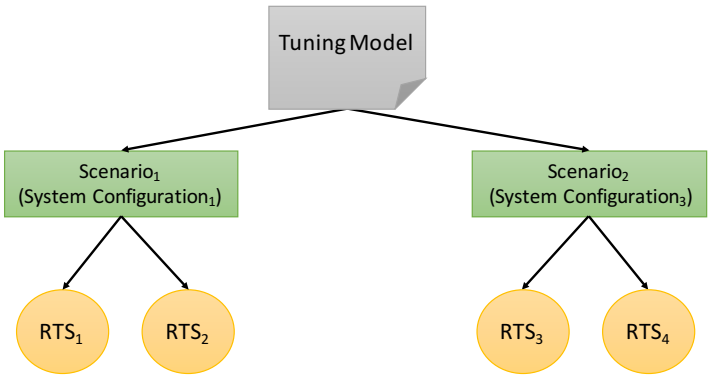

Figure 3: A top-down visual representation of a tuning model.

Figure 2 presents a high-level example where PTF has identified four *rts*'s (rts_1 to rts_4). Each *rts* has a best-found system configuration (*cfg*). In this case it is found that cfg_1 and cfg_2 are identical, and that cfg_3 and cfg_4 are identical. They are hence clustered, and we are ready to create scenarios for the different configurations.

2.2.2 Generation of the Tuning Model

¡¡¡¡¡¡¡ .mine Figure 3 provides a top-down view of a serialised tuning model generated during DTA. In the tuning model, like *rts*'s and configurations, a scenario is also implemented as a map data structure where the keys point to an *rts*, while the values are used to lookup a *cfg* from the map of configurations. In the pre-alpha prototype, the region name was the sole key, while the alpha prototype supports the set of identifiers described above. Both in the pre-alpha and alpha prototype, the selector which maps a *cfg* onto a scenario is implicitly given from this mapping. ======= Figure 3 provides a top-down view of a serialised tuning model generated during DTA. In the tuning model, like *rts*'s and configurations, a scenario is implemented as a map data structure where the keys point to an *rts*, while the values are used to lookup a *cfg* from the map of configurations. In the pre-alpha prototype, the region name was the sole key, while the alpha prototype supports region name, call path, and Score-P user parameters. Both in the pre-alpha and alpha prototype, the selector, which maps a *cfg* onto a scenario, is implicitly given from this mapping. ¿¿¿¿¿¿¿ .theirs

Comment[ZB]: better to be precise

As previously mentioned, our motivation for saving the tuning model as a JSON file is mainly for its human-readability properties. In spite of this, tuning model files for large applications that contain many significant regions/system configurations may impede tasks like debugging. Hence, *Tmviewer*, a tool for visualising the content of a tuning model, has been created. The output of the tool is in content similar to what is shown in Figure 3. Although not a part of the alpha prototype, we have found it to be a great aid in visualising the mapping between scenarios and system configurations.

Comment[KD]: should we include an example of the use of Tmviewer?

Comment[PGK]: Answer:This is the best we can manage for now.

2.3 Features of Runtime Application Tuning (RAT)

The READEX Runtime Library (RRL) implements the second stage of READEX tool-suite; namely RAT. This stage performs application tuning at runtime. The RRL receives the tuning model generated during the DTA phase. During the production run of the application, it automatically switches to the best configurations identified in the tuning model. Moreover, some parts of the RRL are also used during the DTA as mentioned in Section 2.1.

The alpha prototype extends the pre-alpha version. Therefore, we extended the modules already present in the pre-alpha version. Figure 4 shows the current design of the RRL. Modules that have undergone major changes are marked green. Red marks the modules that are developed as part of the READEX tool suite.

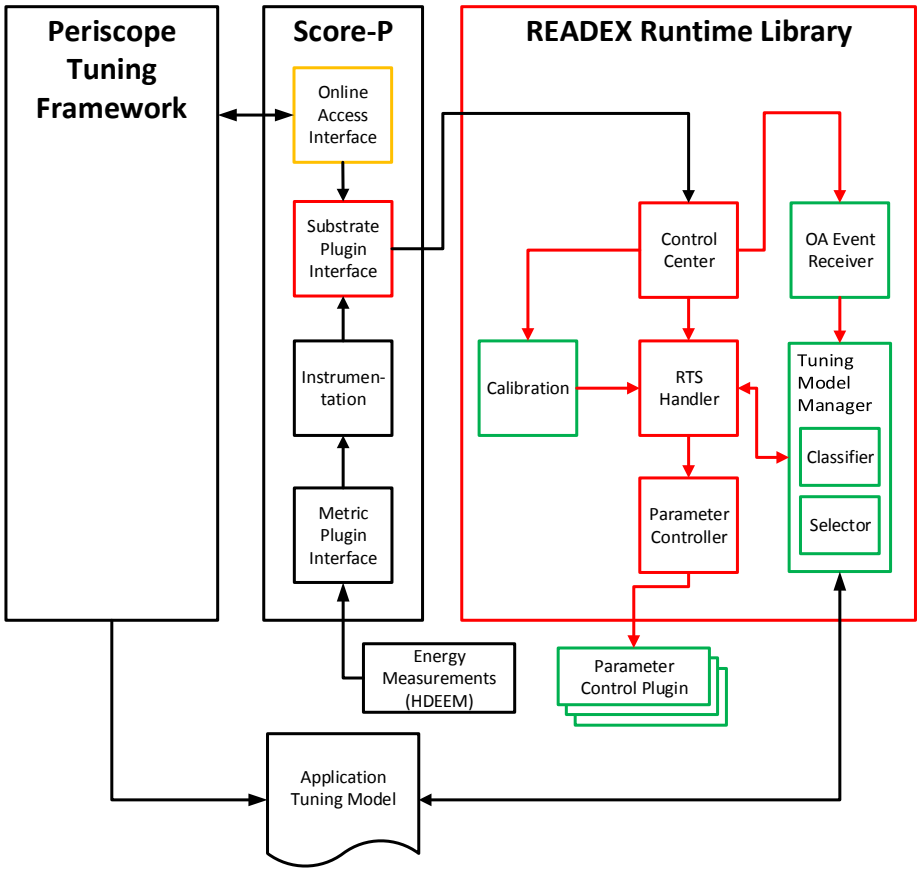

Figure 4: Current architecture of the READEX Runtime Library (RRL). Red: Implementation is part of the extensions made for the READEX tool suite. Green: Implementation has changed since the pre-alpha prototype

The following sections list these changes and give some detailed insight into the changes made. Details about the unchanged modules can be read in the report about the Pre-Alpha Prototype [3].

2.3.1 OA Event Receiver

The OA event receiver provides the interface between the PTF and the RRL and is used during the DTA phase of the READEX tool suite. PTF generates tuning requests which are propagated to Score-P through the so-called online access interface (OA). When a tuning request is received, the OA interface generates an event and passes the tuning request as a JSON formatted string to Score-P. Once the RRL receives the OA event including the JSON string from Score-P, it passes the JSON string (tuning request) to the OA event receiver. In a next step, the OA receiver parses and propagates this information to the Tuning Model Manager described in Section 2.2.

In order to support tuning of different RTSs in one phase, the tuning request parser has been extended. Listing 1 shows the old version of the tuning request and Listing 2 presents the new format.

The new format has a property called “RegionCallpath”. It is used to specify the region where a certain configuration shall be applied. Besides the Region ID, which specifies a certain function, the new tuning request is now also able to take into account additional identifiers. It can be seen that the new format is much more fine grained and allows a more precise setting of tuning parameters for regions.

Listing 1: Format of the OA tuning request V0.1

```
1  {
2      "genericEventType":"tuningRequest",
3      "genericEventTypeVersion":0.1,
4      "data":
5      {
6          "RegionName":<Region name>,
7          "RegionID":<Region id>,
8          "TuningParameters":
9          {
10             "coreFrequency":<Frequency setting>
11         }
12     }
13 }
```

Listing 2: Format of the OA tuning request V0.2

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

2.3.2 Tuning Model Manager

The Tuning Model Manager (TMM) holds the Tuning Model, which holds the configuration for a certain region, and was developed as two modules. One TMM is used during DTA, while the other one is used during the RAT. This allows for a more flexible development. We were able adjust the RAT TMM to support RTS specific tuning, while the DTA TMM was still used during DTA. Once the new OA interface was implemented we were able to switch to the RTS specific tuning for DTA as well. Therefore, we were able to guarantee a working RRL during the development, which allows us to detect errors in an early development stage. This ensured a high code quality.

2.3.3 Parameter Control Plugins

The Parameter Control Plugins (PCP's) perform the configuration of different hardware resources. While the Dynamic Voltage and Frequency Scaling (DVFS) plugin was already available for the pre-alpha prototype, additional PCP's are now supported. The following plugins are now 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].

2.3.4 Calibration

The goal of the calibration is to handle unseen runtime situations (RTS) during the production run of a HPC application. These unseen RTS may consists of already known regions, which have some unknown parameters. Alternatively, these RTS may consist of completely new regions which have totally unknown characteristics. Therefore, the calibration module will use different Machine Learning techniques in order to predict the optimal configuration for an unseen RTS. We outlined an internal concept of the calibration, which will be implemented in the following weeks. The calibration module and the related mechanisms will be part of Deliverable 3.2.

3 Using Alpha Prototype

The complete tool suite built with `gcc/5.3.0` and `bullxmpi/1.2.8.4` (or `intel/2016.2.181` and `intelmpi/5.1.3.181`) can be loaded as follows:

```
module use /projects/p_readex/modules
module load readex/ci_readex_bullxmpi1.2.8.4_gcc5.3.0
```

or

```
module use /projects/p_readex/modules
module load readex/ci_readex_intelmpi5.1.3.181_intel2016.2.181
```

The individual tools built with `gcc/5.3.0` and `bullxmpi/1.2.8.4` (or `intel/2016.2.181` and `intelmpi/5.1.3.181`) can be loaded as follows:

```
module use /projects/p_readex/modules
module load scorep/ci_TRY_READEX_online_access_call_tree_extensions_bullxmpi1.2.8.4_gcc5.3.0
module load ptf/ci_readex_bullxmpi1.2.8.4_gcc5.3.0_slurm_starter_with_scorep
module load readex-rrl/ci_readex_bullxmpi1.2.8.4_gcc5.3.0
module load pcp/ci_pcp_bullxmpi1.2.8.4_gcc5.3.0
```

or

```
module use /projects/p_readex/modules
module load scorep/ci_TRY_READEX_online_access_call_tree_extensions_intelmpi5.1.3.181_intel2016.2.181
module load ptf/ci_readex_intelmpi5.1.3.181_intel2016.2.181_slurm_starter_with_scorep
module load readex-rrl/ci_rrl_intelmpi5.1.3.181_intel2016.2.181
module load pcp/ci_pcp_intelmpi5.1.3.181_intel2016.2.181
```

3.1 Design-Time Analysis

This section describes the Score-P annotation of an application, its dynamism analysis using readex-dyn-detect (with the aid of scorep-autofilter) and its design-time analysis using PTF. The steps are illustrated using the miniMD application as an example.

3.1.1 Annotate phase region

Manually annotate the phase region of the application as shown below:

```
SCOREP_USER_REGION_DEFINE( REGION_HANDLE )
// loop starts
  SCOREP_USER_OA_PHASE_BEGIN( REGION_HANDLE, "PHASE_REGION_NAME", SCOREP_USER_REGION_TYPE_COMMON )
  // loop body (phase region)
  SCOREP_USER_OA_PHASE_END( REGION_HANDLE )
// loop ends
```

Example The `for-loop` body in `Integrate::run()` is annotated as a phase region as shown in Appendix A and is available on Taurus in

```
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/phase_region_integrate.cpp
```

3.1.2 Apply Score-P Auto-filter

1. Build the application with `scorep` without options for manual annotations. Note that Score-P and the application are to be built with the same compiler.
2. Run the application. This will create a `profile.cubex` file in the `scorep-<xyz>` directory at the execution location.
3. Apply the `scorep-autofilter` tool on the `profile.cubex` file as follows:

   ```
   scorep-autofilter -t <region_granularity_threshold_in_ms>
                     -f <filter_file_name_without_extension>
                     <path_to_cubex_file>/profile.cubex
   ```

   This will create a filter file with `.filt` extension.

Repeat steps 2 and 3 to add regions of fine granularity to the filter file, until no more new regions can be added to the filter file. This is necessary to reduce instrumentation overhead, and requires that the environment variable `SCOREP_FILTERING_FILE` be set to the filter file name (including the `.filt` extension) before repeating steps 2 and 3.

Example

1. The miniMD application is built as follows:

   ```
   make openmpi PREP="scorep"
   ```

2. A script to repeat step 2 (run application) and step 3 (apply `scorep-autofilter`) for the miniMD application is available in

   ```
   /projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/do_scorep_autofilter_loop.sh
   ```

   and is presented in Appendix B. The three arguments to this script are: (1) the value for `-t` input to `scorep-autofilter`, (2) the input file name to miniMD, and (3) the number of MPI processes to create for the application.

For different applications, `do_scorep_autofilter_loop.sh` can be reused by updating the line to execute the application. This script requires `do_scorep_autofilter_single.sh` that is present in the same directory. They are to be run from the location with the application's executable and name the filter files as `scorep.filt`.

3.1.3 Apply READEX Dyn-detect

1. Build the application with `scorep --online-access --user --thread=none` for the manually annotated phase region.

2. Run the application with the following environment variables set:

```
export SCOREP_PROFILING_FORMAT=cube_tuple
export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_L3_TCM
export SCOREP_FILTERING_FILE=<filter_file_name_with_extension>
```

   This will create a tupled `profile.cubex` file in the `scorep-<xyz>` directory at the execution location.

3. Apply the `readex-dyn-detect` tool on the `profile.cubex` file as follows:

```
readex-dyn-detect -t <region_granularity_threshold_in_ms>
                  -p <phase_region_name >
                  -c <compute_intensity_variation_threshold_in_percent>
                  -v <execution_time_variation_threshold_in_percent>
                  -w <region_execution_time_weight_wrt_phase_execution_time_in_percent>
                  -f <RADAR_report_file_name>
                  <path_to_cubex_file>/profile.cubex
```

4. The results of `readex-dyn-detect` are summarised in `readex_config.xml` in the execution directory, which is used as an input to PTF. An example of `readex_config.xml` is available in `<PTF_installation_path>/templates/readex_config.xml.default`.

   The `readex_config.xml` file may be manually created from this template and used as input for PTF without applying the Score-P autofilter and READEX Dyn-detect tools.

Example

1. The miniMD application with manually annotated phase region is built for `readex-dyn-detect` as follows:

```
make openmpi PREP="scorep --online-access --user --thread=none"
```

2. When miniMD is run with `in2.data` as its input file and `readex-dyn-detect` is applied on the resulting tupled `profile.cubex` as follows, the function `ForceLJ::compute_halfneigh()` is identified as the significant region.

```
readex-dyn-detect -t 0.001 -p INTEGRATE_RUN_LOOP -c 20 -v 0.01 -w 0.1 scorep-<xyz>/profile.cubex
```

A script to perform steps 2 and 3 for the miniMD application is available in

```
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/do_readex_dyn_detect.sh
```

For different applications, `do_readex_dyn_detect.sh` can be reused by updating the line to execute the application. This is to be run from the location with the application's executable and the filter file name considered to be `scorep.filt`.

3.1.4 Annotate significant regions

Manually annotate the significant regions that are identified by `readex-dyn-detect` (and all their parents in the hierarchy) in the application as shown below:

```
SCOREP_USER_REGION_DEFINE( REGION_HANDLE )
  SCOREP_USER_REGION_BEGIN( REGION_HANDLE, "SIG_REGION_NAME", SCOREP_USER_REGION_TYPE_COMMON )
  // significant region
  SCOREP_USER_REGION_END( REGION_HANDLE )
```

This is an optional step to allow the user to reduce measurement overhead further. With manual instrumentation it can be limited to exactly those significant regions.

Example Manually annotate `ForceLJ::compute_halfneigh()` and its parents `Integrate::run()` and `main()` as significant regions as shown in the following files respectively:

```
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/sig_region_force_lj.cpp
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/sig_region_integrate.cpp
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/sig_region_ljs.cpp
```

3.1.5 Apply PTF

1. Build the application with `scorep --online-access --user --thread=none`.

   Alternatively, build the application with `scorep --online-access --user --thread=none --nocompiler` if the significant regions were manually instrumented to avoid additional measurement overhead from automatic compiler instrumentation.

2. Load the plugin(s) required for energy measurement that are compatible with the Score-P version built for the READEX tools suite.

   For instance, on the Taurus system at TU Dresden, load the `scorep-hdeem sync` plugin for energy measurements and set the following environment variables:

   ```
   export SCOREP_METRIC_PLUGINS=hdeem_sync_plugin
   export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_CONNECTION="INBAND"
   export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_VERBOSE="WARN"
   export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_STATS_TIMEOUT_MS=1000
   ```

3. Use the parameter control plugins compatible with the RRL tuning substrate, and set the following environment variables:

```
export SCOREP_SUBSTRATE_PLUGINS='rrl'
export SCOREP_RRL_PLUGINS=<list_of_parameter_control_plugins_names, eg. cpu_freq_plugin>
```

4. Use and apply the PTF tool on the application as follows:

```
psc_frontend --apprun="<application_run_command>"
             --mpinumprocs=<num_of_mpi_procs>
             --ompnumthreads=<num_of_omp_threads>
             --phase=<phase_region_name>
             --tune=readex_tuning
             --config-file=<readex_config.xml file>
             --info=<max_info_level, eg. 2 to 6>
             --selective-info=<comma_separated_list_of_information_levels, eg. AgentApplComm>
```

This will produce a tuning model in the execution directory under the name `tuning_model.json`.

Example

1. The miniMD application with manually annotated phase region is built for PTF as follows:

```
make openmpi PREP="scorep --online-access --user"
```

2. The `scorep-hdeem sync` for energy measurements for READEX can be loaded and relevant environment variables set as follows:

```
module load scorep-hdeem/sync-2016-07-14-hdeem2.2.2-xmpi-gcc5.3
export SCOREP_METRIC_PLUGINS=hdeem_sync_plugin
export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_CONNECTION="INBAND"
export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_VERBOSE="WARN"
export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_STATS_TIMEOUT_MS=1000
```

3. The relevant environment variables set for tuning the CPU frequency as follows:

```
export SCOREP_SUBSTRATE_PLUGINS='rrl'
module load pcp/ci_pcp_bullxmpi1.2.8.4_gcc5.3.0
export SCOREP_TUNING_PLUGINS=OpenMPTP,cpu_freq_plugin,uncore_freq_plugin;
export SCOREP_RRL_VERBOSE="WARN"
```

4. PTF can be applied on the miniMD application as follows:

```
psc_frontend --apprun="miniMD_openmpi -i in2.data"
             --mpinumprocs=8
             --ompnumthreads=1
             --phase=INTEGRATE_RUN_LOOP
             --tune=readex_tuning
             --config-file=readex_config.xml
             --info=2
             --selective-info=AutotuneAll,AutotunePlugins
```

A script to perform steps 2–5 for the miniMD application is available in

```
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/do_psc_frontend.sh
```

For different applications, `do_psc_frontend.sh` can be reused by updating the command to run the application in `--apprun`. This script is to be run from the location with the application's executable.

3.2 Runtime Application Tuning

This section describes the runtime tuning of an application using the READEX Runtime Library (RRL) using the tuning model generated during the design-time analysis phase, using miniMD as an example.

3.2.1 Tune with RRL

1. Use the RRL and the parameter control plugins built for the READEX toolsuite.
2. Set the environment variables to specify the Score-P substrate plugin as RRL and the parameter control plugins to be used as follows:

```
export SCOREP_SUBSTRATE_PLUGINS='rrl'
export SCOREP_RRL_VERBOSE="WARN"
export SCOREP_RRL_PLUGINS=<list_of_parameter_control_plugins_names, eg. cpu_freq_plugin>
```

3. Set the environment variable to specify the tuning model to be used by the RRL and its location as follows:

```
export SCOREP_RRL_TMM_PATH="<path_to_tuning_model_file>/tuning_model.json"
```

4. Run the application with the tuning module JSON file produced by PTF in the specified location.

Example

1. The environment variables to specify the RRL as the substrate plugin and tune the application can be set as follows:

```
export SCOREP_SUBSTRATE_PLUGINS='rrl'
module load pcp/ci_pcp_bullxmpi1.2.8.4_gcc5.3.0
export SCOREP_TUNING_PLUGINS=OpenMPTP,cpu_freq_plugin,uncore_freq_plugin;
export SCOREP_RRL_VERBOSE="WARN"
```

2. The environment variable to specify the tuning model to be used by the RRL is set as follows:

```
export SCOREP_RRL_TMM_PATH="tuning_model.json"
```

3. The miniMD application can be run with input file `in2.data` as follows:

```
mpirun -np 8 ./miniMD\_openmpi -i in2.data
```

A script to perform these steps for the miniMD application is available in

```
/projects/p_readex/ichec/test_apps/miniMD_readex_toolsuite/do_rrl.sh
```

For different applications, `do_rrl.sh` can be reused by updating the line to run the application. This script is to be run from the location with the application's executable.

4 Summary

The features of the alpha prototype of the READEX tool-suite summarised in this report cover the features that were targeted by M18 as outlined in the deliverable D4.1 [**?**]. This prototype has been evaluated internally using a few benchmark applications (including total FETI solvers such as ESPRESO and PERMON, and a few from the CORAL and Mantevo benchmark suites).

Based on this alpha prototype, a pre-beta version of the READEX tool-suite will be developed by M24 with the following feature updates as described in D4.1 [**?**]:

- Design-Time Analysis (DTA)
  - Inter-phase dynamism analysis using phase identifiers, in addition to previously supported intra-phase dynamism analysis using region identifiers.
  - A final set of READEX tuning plugins to support tuning for multiple aspects.
  - A preliminary version of the domain knowledge specification to leverage user domain knowledge.
  - A phase-aware tuning model incorporating phase identifiers for scenario identification and more complex selectors.
- Runtime Application Tuning (RAT)
  - A final set of READEX tuning plugins to execute tuning for multiple aspects.
  - Scenario detection to handle phase scenarios, identified within the phase-aware tuning model.
  - For scenario switching, a global decision making mechanism will be implemented to support more complex selectors and global synchronisation for tuning actions.

Comment[ZB]: May be another interesting feature that can be added easily if not already available is measuring the energy consumption of the entire application and output it in a file or on standard output. It'll enable quick comparison of several READEX results.

References

[1] Openmp application program interface. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[2] Andreas Gocht, Zakaria Bendifallah, Umbreen Sabir Mian, and Othman Bouizi. D1.1: Hardware and system-software tuning plugins. *READEX WP1 Report*, 2016.

[3] Andreas Gocht, Umbreen Sabir Mian, Michael Lysaght, Venkatesh Kannan, Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy, Per Gunnar Kjeldsberg, and Nico Reissmann. Pre-alpha prototype report. *READEX Pre-Alpha Report*, 2016.

[4] Robert Schöne and Daniel Molka. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - Research and Development*, 29(3-4), 2014. DOI: 10.1007/s00450-013-0243-7.

A MiniMD Phase Region Annotation

Listing 3: MiniMD phase region annotation in lines 14, 17 and 113.

```
1  void Integrate::run(Atom &atom, Force* force, Neighbor &neighbor,
2                      Comm &comm, Thermo &thermo, Timer &timer)
3  {
4    int i, n;
5    comm.timer = &timer;
6    timer.array[TIME_TEST] = 0.0;
7    int check_safeexchange = comm.check_safeexchange;
8
9    mass = atom.mass;
10   dtforce = dtforce / mass;
11   #pragma omp parallel private(i,n)
12   {
13
14     SCOREP_USER_REGION_DEFINE(R1)
15     for(n = 0; n < ntimes; n++)
16     {
17       SCOREP_USER_OA_PHASE_BEGIN(R1, "INTEGRATE_RUN_LOOP", 2)
18
19       #pragma omp barrier
20       x = &atom.x[0][0];
21       v = &atom.v[0][0];
22       f = &atom.f[0][0];
23       xold = &atom.xold[0][0];
24       nlocal = atom.nlocal;
25
26        initialIntegrate();
27
28       #pragma omp barrier
29       #pragma omp master
30       timer.stamp();
31
32       if((n + 1) % neighbor.every)
33       {
34         #pragma omp barrier
35         comm.communicate(atom);
36         #pragma omp master
37         timer.stamp(TIME_COMM);
38         #pragma omp barrier
39       }
40       else
41       {
42         {
43           if(check_safeexchange)
44           {
45             #pragma omp master
```

```
46              {
47                double d_max = 0;
48                for(i = 0; i < atom.nlocal; i++)
49                {
50                  double dx = (x[3 * i + 0] - xold[3 * i + 0]);
51                  if(dx > atom.box.xprd) dx -= atom.box.xprd;
52                  if(dx < -atom.box.xprd) dx += atom.box.xprd;
53                  double dy = (x[3 * i + 1] - xold[3 * i + 1]);
54                  if(dy > atom.box.yprd) dy -= atom.box.yprd;
55                  if(dy < -atom.box.yprd) dy += atom.box.yprd;
56                  double dz = (x[3 * i + 2] - xold[3 * i + 2]);
57                  if(dz > atom.box.zprd) dz -= atom.box.zprd;
58                  if(dz < -atom.box.zprd) dz += atom.box.zprd;
59                  double d = dx * dx + dy * dy + dz * dz;
60                  if(d > d_max) d_max = d;
61                }
62                d_max = sqrt(d_max);
63                if((d_max > atom.box.xhi - atom.box.xlo) || (d_max > atom.box.yhi -
                      atom.box.ylo) || (d_max > atom.box.zhi - atom.box.zlo))
64                  printf("Warning: Atoms move further than your subdomain size, which will
                        eventually cause lost atoms.\n"
65                  "Increase reneighboring frequency or choose a different processor grid\n"
66                  "Maximum move distance: %lf; Subdomain dimensions: %lf %lf %lf\n",
67                  d_max, atom.box.xhi - atom.box.xlo, atom.box.yhi - atom.box.ylo,
                        atom.box.zhi - atom.box.zlo);
68              }
69            }
70
71            #pragma omp master
72            timer.stamp_extra_start();
73            comm.exchange(atom);
74            comm.borders(atom);
75            #pragma omp master
76            {
77              timer.stamp_extra_stop(TIME_TEST);
78              timer.stamp(TIME_COMM);
79            }
80            if(check_safeexchange)
81              for(int i = 0; i < 3 * atom.nlocal; i++) atom.xold[i] = atom.x[i];
82          }
83          #pragma omp barrier
84          neighbor.build(atom);
85
86          #pragma omp barrier
87          #pragma omp master
88          timer.stamp(TIME_NEIGH);
89        }
90        force->evflag = (n + 1) % thermo.nstat == 0;
91        force->compute(atom, neighbor, comm, comm.me);
```

```
92
93          #pragma omp master
94          timer.stamp(TIME_FORCE);
95
96          if (neighbor.halfneigh && neighbor.ghost_newton)
97          {
98            comm.reverse_communicate(atom);
99
100           #pragma omp master
101           timer.stamp(TIME_COMM);
102         }
103         v = &atom.v[0][0];
104         f = &atom.f[0][0];
105         nlocal = atom.nlocal;
106
107         #pragma omp barrier
108          finalIntegrate ();
109
110         #pragma omp barrier
111         if (thermo.nstat) thermo.compute(n + 1, atom, neighbor, force, timer, comm);
112
113       SCOREP_USER_OA_PHASE_END(R1)
114       }
115   } //end OpenMP parallel
116 }
```

B Scripts to Apply Scorep-Autofilter

Listing 4: do_scorep_autofilter_loop.sh

```
1  #!/bin/sh
2
3  export SCOREP_FILTERING_FILE=scorep.filt
4  rm -rf scorep-*
5  rm -f old_scorep.filt
6  echo "" > scorep.filt
7  result=1
8  while [ $result != 0 ]; do
9    # run the application.. update this for different applications
10   mpirun -np $3 ./miniMD_openmpi -i $2 1>/dev/null
11   echo "run miniMD done."
12   sh -l do_scorep_autofilter_single.sh $1
13   result=$?
14   echo "scorep_autofilter_singe done ($result)."
15 done
16 echo "scorep_autofilter_loop done."
```

Listing 5: do_scorep_autofilter_single.sh

```
#!/bin/sh

cp scorep.filt old_scorep.filt

scorep-autofilter -t $1 -f scorep scorep-*/profile.cubex 1>/dev/null
echo "#############"
cat scorep.filt
echo "#############"

rm -rf scorep-*
commString=$(comm -1 -2 scorep.filt old_scorep.filt)
if [ "$commString" == "" ]; then
  cp scorep.filt old_scorep.filt
  exit 1
else
  filtFileBeginString="SCOREP_REGION_NAMES_BEGIN"
  filtFileEndString="SCOREP_REGION_NAMES_END"
  filtFileExcludeString="EXCLUDE"
  sort scorep.filt > sorted_scorep.filt
  sort old_scorep.filt > sorted_old_scorep.filt
  commString=$(comm -2 -3 sorted_scorep.filt sorted_old_scorep.filt)
  if [ "$commString" != "" ]; then
    echo $filtFileBeginString > new_scorep.filt
    echo $filtFileExcludeString >> new_scorep.filt
    comm -2 -3 sorted_scorep.filt sorted_old_scorep.filt > comm1.txt
    exec < "old_scorep.filt"
    while read line; do
      if [ "$line" != "$filtFileEndString" ] && [ "$line" != "$filtFileBeginString" ] && [ "$line"
          != "$filtFileExcludeString" ]; then
        echo $line >> new_scorep.filt
      fi
    done
    cat comm1.txt >> new_scorep.filt
    echo $filtFileEndString >> new_scorep.filt
    mv new_scorep.filt scorep.filt
    rm -f sorted_scorep.filt sorted_old_scorep.filt comm1.txt
    exit 1 # new entries added to filter file
  else
    mv old_scorep.filt scorep.filt
    rm -f sorted_scorep.filt sorted_old_scorep.filt comm1.txt
    exit 0 # no new entries added to filter file
  fi
fi
```