Horizon 2020
European Union funding
for Research & Innovation

GA no. 671657

READEX
Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing

D1.2
Final Tuning Plugins

Document type:	Other
Dissemination level:	Public
Work package:	WP1
Editor:	Umbreen Sabir Mian (TUD), Zakaria Bendifallah (Intel)
Contributing partners:	TUD, Intel
Reviewer:	Kai Diethelm (GNS) Per Gunnar Kjeldsberg (NTNU)
Version:	3.0

Document history

Version	Date	Author/Editor	Description
0.1	15/06/17	Umbreen Sabir Mian (TUD)	1st TOC draft
0.2	09/06/17	Umbreen Sabir Mian (TUD)	Added reviewers and writing responsibilities
0.3	03/07/17	Umbreen Sabir Mian (TUD)	Initial version of Sections Hardware parameters and System Software Parameters
0.4	03/07/17	Umbreen Sabir Mian (TUD)	Initial version of Sections Introduction, Integration in the READEX tool suite
0.5	03/07/17	Zakaria Bendifallah(Intel)	Initial version of Section Application Parameters
0.6	04/07/17	Umbreen Sabir Mian (TUD)	Initial version of Sections Executive summary, and Summary
0.7	05/07/17	Zakaria Bendifallah(Intel)	Improved Section Application Parameters
0.8	25/07/17	Umbreen Sabir Mian (TUD)	Improved Sections Hardware parameters, System Software Parameters
0.9	01/08/17	Umbreen Sabir Mian (TUD)	Improved Sections Executive summary, Introduction and Summary
1.0	07/08/17	Umbreen Sabir Mian (TUD)	First draft ready for review
1.1	17/08/17	Umbreen Sabir Mian (TUD)	Addressed review comments in Executive summary Sections 1, 2, 3, 4, 5 and 7
1.2	18/08/17	Zakaria Bendifallah(Intel)	Addressed comments on Section 6
2.0	20/08/17	Umbreen Sabir Mian (TUD)	Second draft ready for review
2.1	25/08/17	Umbreen Sabir Mian (TUD)	Diethelm's second review comments addressed in Executive summary, Sections 1-5 and 7
2.2	29/08/17	Umbreen Sabir Mian (TUD)	Kjeldsberg's second review comments addressed in Executive summary, Sections 1, 2, 3, 4, 5 and 7
2.3	29/08/17	Zakaria Bendifallah(Intel)	Second review comments on Section 6 addressed
3.0	30/08/17	Umbreen Sabir Mian (TUD)	Final Version ready for submission

Executive Summary

The objectives of Work Package 1 (WP1) “Tuning parameters” are to determine parameters that can be tuned and to develop and implement concepts for their integration into the READEX tool suite. The tuning parameters are split in three different levels of the HPC stack: Hardware, Runtime System, and Application.

A multitude of hardware and runtime system parameters have already been presented in Deliverable 1.1 [4] and the results of tuning these parameters obtained using the then available READEX tool suite have also been described there. In this deliverable, we present the current state of the tuning plugins and the Parameter Control Plugins (PCPS) shortlisted in Deliverable 1.1.

In addition, Deliverable 1.1 only included some basic concepts for application parameter tuning. At this stage of the project, the implementation and testing of the application parameter tuning component is finished. So the main focus of this deliverable is to present the research findings and implementation details for application parameter tuning.

After a short introduction, this deliverable starts with an outline of the current READEX tool suite. We describe in which places we have to hook in to integrate the different tuning parameters as well as the basic concepts of the READEX tool suite.

Since all the details about our initial assumptions as well as our concrete findings about the hardware and runtime system parameters have already been presented in Deliverable 1.1, we will skip all those details in this deliverable. Our results showed that the processor core and uncore frequencies are the most promising hardware tuning parameters. Moreover, the number of threads used for OpenMP computations as well as the scheduler policy and the scheduler chunk size are promising parameters for runtime system parameters. For MPI parameters, results were presented for `MPIR_CVARS_REDUCE_SHORT_MSG_SIZE`. The results were not very appealing for tuning but we implemented a tuning and a control plugin because each MPI implementation have different parameters available. Some of these parameters might be interesting for the user of READEX tool suite. The control plugin provides a C++ interface which user can use to implement other MPI parameters of interest.

Finally, we describe our formalization and implementation for the application parameters. We have developed a user API for application tuning parameters which can be used by the user at runtime to declare new application parameters and change the values of the already declared application parameters. We show with an example how the user can benefit from application specific parameter tuning by using the Application Tuning Parameter (ATP) library.

Contents

1 Introduction

The READEX project focuses on the dynamic tuning of applications in order to improve their energy efficiency. Therefore, we need different parameters that have a noticeable impact on the energy efficiency of programs. In this Deliverable, we present the tuning plugins and the Parameter Control Plugins (PCPs) for the parameters that have been outlined in Deliverable 1.1 [4] and that can be tuned using the READEX tool suite.

For energy efficiency tuning, each hardware and software parameter needs a corresponding PCP which will allow its integration into the READEX tool suite. Therefore we will start in Section 2 with a short description of the READEX tool suite. A brief outline of how these plugins are controlled and where these plugins have to be integrated will be presented. The interface between the PCPs and the READEX Runtime Library (RRL) has been changed in the current version of the READEX tool suite and its details will be given in this deliverable. For tuning the application parameters, a separate component — the so-called ATP Plugin — has been developed. In Section 2, we also describe how the ATP Plugin interacts with the rest of the tool suite.

We start in Section 3 with the details of the interface of PCPs, also outlining the implementation details. The following Sections 4, 5 and 6 present tuning details for hardware parameters, system software parameters, and application-level parameters, respectively.

Finally, we will summarize all our findings about tuning parameters and the READEX tool suite's capabilities for tuning the presented tuning parameters.

1.1 Experiment Platform

All experiments for the tuning parameters are conducted on the Taurus system installed at TU Dresden. This system comprises of 1456 nodes each containing two 12-core Intel Xeon CPUs E5-2680 v3 (Intel Haswell processor family) running with a default frequency of 2.50 GHz. The nodes contain between 64 and 256 GB of memory.

All nodes are equipped with the HDEEM energy measurement system [6]. This system allowed us to do reasonably accurate measurements as the measurement error is 2% for measurements of the whole node [2].

2 Integration in the READEX Tool Suite

All tuning plugins and parameters that we describe in the following sections have been integrated into the READEX tool suite. Therefore, we start with a small overview of the READEX tool suite stack.

Figure 1 outlines the interaction between the Periscope Tuning Framework (PTF), the Score-P measurement infrastructure, the RRL, the PCPs and the ATP plugin.

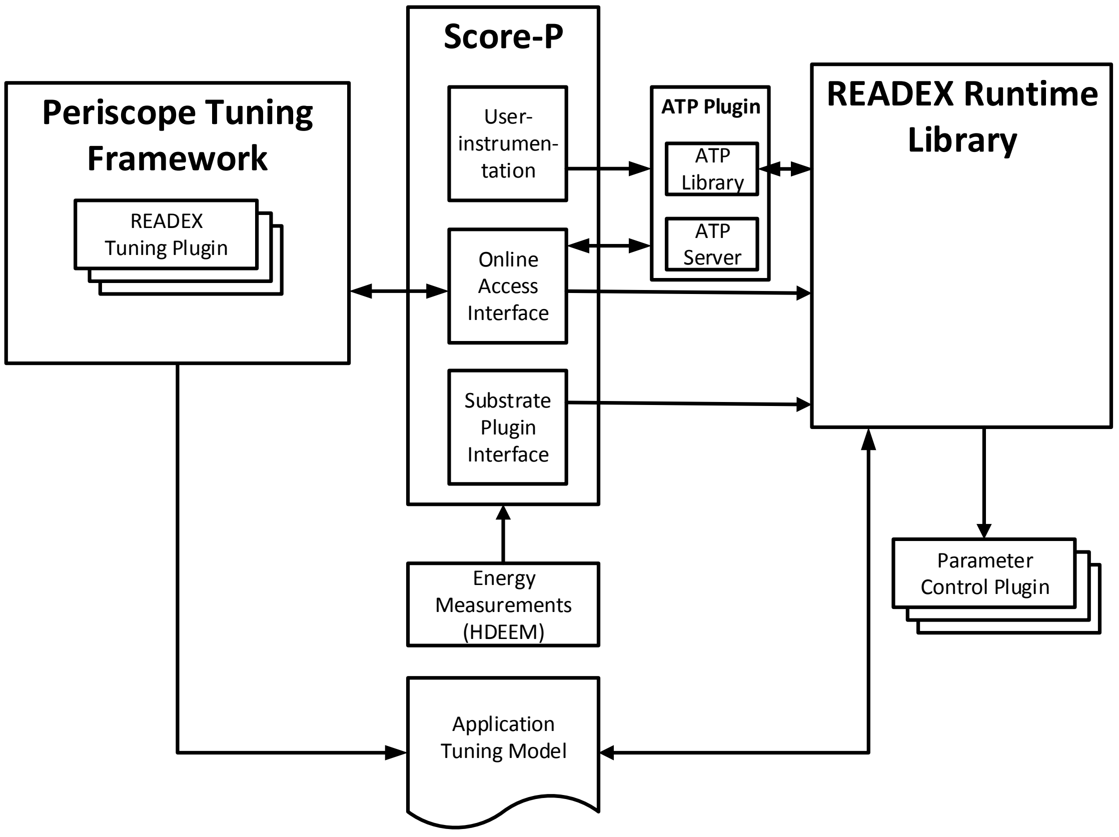

Figure 1: Overview of the READEX tool suite. The graph shows the information flow between all the tools. The "Parameter Control Plugins" are the plugins that change the hardware and software tuning parameters. The "ATP Plugin" handles the tuning of application parameters.

The RRL controls the PCPs. These plugins are responsible for changing different hardware or software parameters.

The RRL methodology is applied in two phases, Design Time Analysis (DTA) and Runtime Application Tuning (RAT).

During DTA, the RRL receives its tuning commands through the Score-P Online Access (OA) interface. The interface in turn is connected to PTF. A tuning command contains the name of the parameter to change, the parameter value and the program region where it shall be applied. PTF obtains the information about the available parameters and the parameter values from the READEX Tuning Plugin. Once PTF finds an optimal configuration, it saves this configuration in the so called application tuning model.

During RAT, the RRL requests the optimal configuration from the application tuning model and passes it to the PCPs.

The ATP plugin handles the tuning of application parameters. It consists of two components: an ATP server that provides the interface between the application and PTF through the Score-P OA interface and an API named as ATP library that includes API calls. The users have to use ATP library calls within the application to declare and get ATPs.

In DTA, the ATP library creates an ATP description file using the API calls inserted by the user in the application. The ATP server reads the ATP description file and records the application parameters details. It receives and responds to the requests coming from PTF via the Score-P Online Access Interface. PTF requests include queries on application parameter details such as the list of parameters and the values for these parameters. Once PTF gets these details, it sends tuning commands for application parameters to RRL through the Score-P interface in the same way as for software and hardware parameters.

At RAT, the ATP library requests the optimal configuration from the RRL. The RRL gets the optimal configuration from the application tuning model and passes it to the ATP library which further passes the received configuration to the application.

In a future version of the READEX tool suite, we will implement a calibration mechanism which will be activated during runtime. If a runtime situation occurs that has not been seen during design time, the calibration mechanism will search for an optimal configuration of this scenario.

The different parts of the software are maintained in three different repositories. We use a modified version of Score-P which we will merge to the official Score-P once our changes are final. Our Score-P version can be accessed from the non-public Score-P SVN project archive hosted by TU Dresden. PTF together with the tuning plugins is located at a non-public Git project archive hosted by TU München. The RRL, the PCPs and the ATP plugin are located at a non-public Git project archive hosted by TU Chemnitz. Access to these archives is granted after requests to responsible project partners. This way of getting access to the READEX tool is a temporary solution and is valid during the development period. After the release of first public version, the changes made in Score-P will be available in the public version of Score-P. The same is true for PTF. The RRL, ATP plugin and PCPs will be available as one module with easy to install configuration.

3 PCP Interface

PCPs are used for changing different hardware and software parameter values. A common interface is provided for each PCP, which contains the name of a parameter, a set function, an unset function, and a get current configuration function. RRL communicates with PCPs using this interface to set and unset the parameter values and also to retrieve the current configuration for the available parameters.

Listing 1: PCP Interface

```c
char *name;                          // tuning parameter name
int (*current_config)();             // get current configuration
void (*enter_region_set_config)(int); // set configuration at enter
    region event
void (*exit_region_set_config)(int); // set configuration at exit region
    event
```

Listing 1 shows the interface provided by each PCP. The fields in the listing are explained below:

- `name` simply refers to the name of the tuning parameter for which the PCP applies tuning.
- The function `current_config()` returns the current configuration of the tuning parameters.
- `enter_region_set_config(int)`: This method is called at enter region event to set the value of the tuning parameter to the value passed as argument.
- `exit_region_set_config(int)`: This method is called at exit region event to reset the value back to what it was before entering that particular region. The old configuration is stored internally and is passed as an argument here. This functionality is optional and the user can turn it off using the environment variable `SCOREP_RRL_CHECK_IF_RESET`. By default, the reset functionality is enabled.

3.1 Workflow

The overall workflow of the hardware and software parameter tuning during DTA and RAT is as follows:

As a first step, the application is compiled with Score-P.

- **During DTA**:
  - PTF launches the application executable. A READEX configuration file is given as input to PTF. For details about the READEX configuration file, please refer to Deliverable 4.2 [5].

  - Inside PTF, the READEX Tuning Plugin has been implemented. It supports multiple objective functions, multiple search algorithms and multiple tuning parameters. Currently, the READEX tuning plugin supports the following tuning parameters: core frequency, uncore frequency, energy performance bias and the number of OpenMP threads. The configuration of these 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. The plugin uses the search algorithm specified in the READEX configuration file to generate different configurations for each runtime situation. These configurations are forwarded to RRL as tuning requests.
  - The RRL receives tuning requests from PTF through the Score-P OA interface. The tuning request contains the name of the parameter to change, the parameter value and the program region where it shall be applied. The details about the format of the tuning request can be found in Deliverable 4.2 [5].
  - Each tuning request is stored by the RRL. During DTA execution of the application, the RRL calls the respective PCP to set the requested configurations. Details can be found in Deliverable 3.1 [3].
  - The READEX Tuning Plugin then computes the best configuration and saves this configuration into the application tuning model.

- **During RAT**:
  - Upon receiving an enter region notification from Score-P during the application run, the RRL gets the best configuration for that particular runtime situation from the application tuning model.
  - The configuration received is then passed to the PCPs where switching of parameter value is performed.
  - If the reset functionality is enabled, the RRL sets the parameter value back to previous value on receiving an exit region event in the same way as set at enter region event.

3.2 Usage

To use the tuning functionalities of the READEX toolsuite, the following steps have to be followed.

- DTA is applied using the PTF command `psc_frontend`. To use the READEX Tuning Plugin, the option `--tune = readex_tuning` has to be set for the `psc_frontend` command.
- Both for DTA and RAT, the required PCPs need to be loaded using the environment variable "SCOREP_RRL_PLUGINS" provided by RRL. Multiple PCPs can be loaded simultaneosly by giving a comma seperated list of PCP names to the above mentioned environemnt variable.

4 Hardware Parameters

The most relevant hardware parameters that we consider are processor related parameters. This is mainly due to the fact that the processor has the highest power consumption in a computer system. In this section, we will outline the PCPs for each of the hardware tuning parameters that we implemented as part of the READEX project.

4.1 CPU Frequency Scaling and Uncore Frequency Scaling (UFS)

We implemented and tested a parameter control plugin and a tuning plugin for both the core and the uncore frequencies. The UFS plugin uses the x86_adapt library [10] to set the relevant machine specific register (MSR) of the processor.

Both plugins allow the user to specify a frequency range that should be searched as well as the size of each step.

To load the control plugins for setting CPU frequency and uncore frequency, the environment variable `SCOREP_RRL_PLUGINS` value should include `cpu_freq_plugin` and `uncore_freq_plugin` respectively.

4.2 Energy Performance Bias

A control and a tuning plugin has been implemented for the Energy Performance Bias (EPB). Both plugins are similar to the UFS plugin. They use the x86_adapt library [10] as well in order to set the related MSR from the userspace. As we want to be able to achieve most energy savings, we decided to make all 16 settings for the EPB accessible to the user.

Therefore, to use the plugin the user has to specify an EPB range that should be searched as well as the size of each step.

In order to load the control plugin for setting energy performance bias frequency, the environment variable `SCOREP_RRL_PLUGINS` value should include `epb_plugin`.

5 System Software Parameters

As mentioned in Deliverable 1.1 [4], we investigated both MPI and OpenMP programming models to target distributed memory parallelism and thread-level parallelism, respectively.

Further we outline the tuning and control plugins implemented in the READEX project for tuning both MPI and OpenMP parameters.

5.1 MPI

The MPI 3.0 standard has introduced the so-called MPI-T interface that allows MPI implementations to offer a set of parameters that can be read and written by the user during runtime. These are so called CVars. A tuning and a control plugin has been implemented for the MPI-T CVars. The tuning plugin allows to specify the CVar that the user wants to change as well as the value to set. Unfortunately, the set of available CVar parameters can vary between different MPI implementations or even between different versions of the same implementation. Therefore, we decided to design a user interface that is as flexible as possible. In the current implementation of the plugin, the user has to specify each parameter he or she would like to tune and has to implement this CVar in the parameter control plugin.

The control plugin for setting MPI-T CVars can be loaded by setting the environment variable `SCOREP_RRL_PLUGINS` to `mpit_plugin`.

5.2 OpenMP

We implemented a parameter control plugin and a tuning plugin to tune OpenMP parameters. The tuning plugin allows the user to specify the number of threads that he/she wants to search.

Moreover, the plugin itself changes the scheduler policy according to the available policies specified in the OpenMP specification [1]. Here the scheduler policy defines how OpenMP schedules loop iterations. Finally, the tuning plugin is able to change the chunk size which is associated to the scheduler. Each thread will take a set number of iterations, called a "chunk", execute it, and then be assigned a new chunk when it is done. The chunk size specifies the number of iterations in a chunk which each thread gets assigned.

In order to load the control plugin for setting OpenMP parameters (number of threads, scheduler policy and chunk size), the environment variable `SCOREP_RRL_PLUGINS` has to be set to `OpenMPTP`.

6 Application Parameters

The idea behind READEX is to exploit the dynamism available inside the application and use it to reduce its energy consumption. The approach to exploit this dynamicity can be subdivided into two main categories:

1. Code-agnostic approach, which includes hardware and system level parameters discussed in Sections 4 and 5 and
2. Code-aware approach, which is discussed in the current section.

In the first category, no advanced knowledge on the application code itself, what it does or what algorithms are used, is necessary. Profiling techniques on the application are sufficient to detect and define the hints to exploit dynamism. On the other hand, the code-aware approach involves both automatic exploration and the knowledge of the application developer. In this case, the developer can give the tool hints to find spots where dynamism can be exploited.

A few examples of the spots that can be exploited for tuning are the following possibilities:

- To choose different code paths corresponds to cases where the developer identifies more than one way to solve a problem. Examples are use of different types of iterative and direct solvers, and alternative preprocessing of stiffness and of coarse problem (CP) matrices. This results in different pieces of code to choose from. What the user does not know is what their characteristics are and how they can be exploited for energy tuning.
- To control problem decomposition (number of sub-domains, cache blocking, . . .).
- To give hints on data structures (array length, access strides, indirections, . . .).

6.1 Research Summary

In order to assess the potential advantage READEX would gain by integrating the support of application level parameters, we based our research on applications which integrated in their code the possibility to choose between different algorithms/solutions, hence we based most of our experiments on the ESPRESO library [7].

The ESPRESO library is a combination of Finite Element (FEM) tools and domain decomposition based Finite Element Tearing and Interconnect (FETI) solvers. The FETI solver contains a projected conjugate gradient (PCG) solver, and therefore its convergence can be improved by several preconditioners. The computational complexity of different preconditioners varies from basic vector scaling (weight function) via sparse-matrix vector multiplication with different numbers of non-zeros (lumped, light-Dirichlet) to dense matrix-vector multiplication (Dirichlet). Using a simplified approximation, we can state that from the preconditioners listed above, the more computationally demanding the preconditioner is, the more numerically efficient it is, i.e. the more it reduces the number of iterations required to solve the problem. In ESPRESO, we can dynamically switch between any of these during runtime. If no preconditioner is used, one iteration contains an action of a FETI operator (cost is 30.9 J and 0.12 s) and an application of a projector (cost is 0.7 J and 0.005 s). If a preconditioner is used, each iteration contains one more projector application in addition to the preconditioner action.

We evaluated the preconditioners on a structural mechanics (linear elasticity) problem with 2.3 million unknowns on a single compute node using 24 MPI processes. The results, see Table 1, show that using the light Dirichlet preconditioner is the fastet way to reach the solution despite the fact that it needs more iterations than the Dirichlet preconditioner. The light Dirichlet preconditioner required only 5.46 s and thus saved 15.9 s and 4 091.5 J in comparison to solving the problem without any preconditioner.

Preconditioner	**# iterations**	**1 iteration**		**Solution**	
none	172	125 ms	31.6 J	21.36 s	5 501.31 J
Weight function	100	130+2 ms	32.3+0.53 J	12.89 s	3 284.07 J
Lumped	45	130+10 ms	32.3+3.86 J	6.32 s	1 636.11 J
Light Dirichlet	39	130+10 ms	32.3+3.74 J	5.46 s	1 409.82 J
Dirichlet	30	130+80 ms	32.3+20.62 J	6.34 s	1 594.50 J

Table 1: ESPRESO preconditioners comparison for runtime and energy consumption. The table contains (i) single iteration evaluation including baseline (FETI operator and 2× projector) + resources spent by the preconditioner, (ii) overall FETI solver evaluation considering the different number of solver iterations.

The above-mentioned results provide a clear motivation to integrate a semi-automatic approach in handling application parameters which would be based on two major steps:

1. Light and flexible annotation of the application code by the developer to declare application variables. This has been done through a READEX provided API.
2. Automatic exploration of the different possible values for each application parameter, as well as assessment of the best choices by the READEX runtime system.

6.2 Implementation

The ATP plugin designed to exploit application parameters differs in its implementation and working from other (hardware and system level) plugins. Two particularities of ATPs are at the base of this difference:

- Application tuning parameters are tied to the application workflow itself, and thus cannot be changed on the fly at any point of the execution like core frequency for example. Instead, their value can only be changed at specific points in the application (when application control needs to take a decision based on the parameter variable).

- Depending on the parameter, the possible values of an application tuning parameter are likely to be determined during application execution.

An application tuning parameters plugin has been implemented as a standalone component in READEX. The component is composed of two parts: 1) ATP library, and 2) ATP server. Figure 2 illustrates how the component is connected to the two major components of the READEX tool suite, that is: PTF and Score-P.

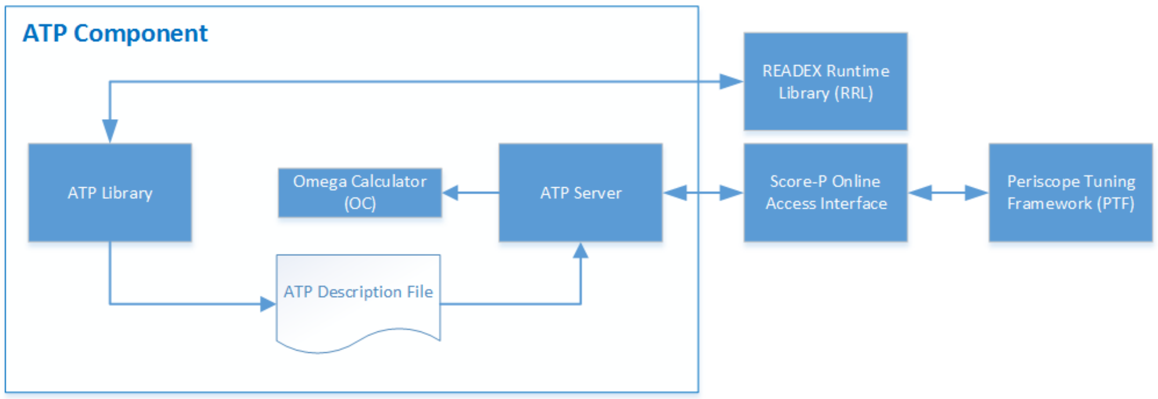
ATP Component
READEX Runtime Library (RRL)
ATP Library
Omega Calculator (OC)
ATP Server
Score-P Online Access Interface
Periscope Tuning Framework (PTF)
ATP Description File

Figure 2: ATP component composition and relations with other READEX components.

6.2.1 ATP library

The ATP library is an annotation library which provides an annotation API to the application developer. The API allows to

- mark the chosen variables as application parameters, and also specify the following details:
  - **types**: currently only tuning variables (to which we refer as parameters) of integer type are handled by the component.
  - **values**: the possible values that a variable can take can be declared either as a range with the format [min values, max value, increment], or as an enumeration of values in the form of an array.
  - **dependencies**: tuning variables in the application may depend on each other, in which case not all combination of values are valid. Therefore, the ATP library enables to express the dependencies between variables in the form of logical constraints in order to be able to generate the valid combination of values. The constraints handled so far in the ATP component are those that can be expressed

by affine functions, an example of such constraints would be: $p1 >= p2 * 4$ where *p1* and *p2* are parameter names.

- Provide to the RRL information of the parameters and their location.
- Assign the parameter values generated by PTF and prepared by RRL to the right parameter.
- Generate an *ATP description file* of all the parameters declared during the application run.

More details on parameter types and constraints handled by the ATP library can be found in Deliverable 4.5.

6.2.2 ATP description file

The ATP description file is a JSON file generated by the ATP library at the end of the first phase of application execution, once all the parameter declaration functions have been encountered. The file is meant to be exploited by the ATP server.

The ATP description file has the structure described below. A detailed description of each of its blocks is given in Table 2.

```
{
  "dom name": {
    "parameters":{
      "param name":{"range":[...], "type":"...", "default":"..." },
      ...
    },

    "constraints":{
      "constr name":{"expr":"..."},
      ...
    },

    "exploration":[...]
  }
}
```

6.2.3 ATP server

The ATP server constitutes the part which communicates with PTF by providing responses to its requests. The ATP server includes the following features:

No.	Block name	Block description
1	Domain block	Is at the highest level and contains all the other blocks. The domain block serves as a container for a set of variables, which have dependences between them. It has mainly an organizational purpose where it allows the developer to reuse variable names by putting them in different contexts and avoid ambiguity.
2	Parameters block	Holds all the parameters declared in the wrapping container with their details. The block is identified through the `parameters` keyword. Each parameter has an identifier name followed by its description: range: indicates that the parameter has a range of values The minimum value, maximum value and increment are given in the form of an array. type: indicates how to interprete the range field, either as a range (min,max,increment) or as an enumeration of values default: indicates the default values to assign to the parameter in case none is specified at runtime.
3	Constraints block	Holds the listing of constraints which may exist between the parameters declared in the parameters block. The block is identified through the `constraints` keyword. Each constraint has an identifier name followed by its description: expr: indicates the relational expression between two or more parameters in the same domain.
5	Exploration block	Holds a list of search methods to apply on the domain. The methods are applied in the given order in the form of an array. The block is optional. If it is not given, the tuning system decides automatically on the methods to use.

Table 2: Description of the different blocks in the ATP description file.

- Reads the contents of the *ATP description file* and records the details on application parameters.
- Receives and responds to requests from PTF on application parameters details such as the list of parameters and the list of valid points for the parameters.
- Queries a third party constraints solver software called *The Omega Calculator* [9] in order to obtain the tuples of valid points for ATPs with constraints between them. The software is composed of a library *Omega Library* which constitues the core of the solver as well as a text interpreter to query the library. The software is registered under the BSD licence and the source code can be downloaded freely from Github [8]. One big advantage of using the Omega Calculator is the extra small computational time needed to solve the affine function based constraints, which makes it fit for use to solve the constraints at runtime.

6.3 Integration

In order to provide the application with access to API calls, the ATP library needs to be linked with the application at compile time. Also, as the RRL uses the ATP library API to assign parameter values, the library needs to be linked with the RRL as well. On the other hand, the ATP server is launched by PTF at execution startup and makes use of the ATP description file that the ATP library generates. The integration of the ATP component in READEX is the same in both DTA and RAT, the only difference is in the behaviour of the ATP library where in DTA it both process parameter declaration and value assignment whereas in RAT parameter declaration is disabled and only value assignment remains active.

6.4 Workflow

The overall workflow of the ATP component can be summarized into the following steps:

- The application developer includes the file `atplib.h` in the source code and annotates it with declarations of application parameters. Following this, the code is compiled with Score-P and linked with the ATP library. RRL is also linked to the ATP library.
- **During DTA**
  - PTF launches the application executable and the ATP server.
  - During the first phase of the application, the ATP library records all application parameter details declared through the API. At the end of the phase, the library exports the details into an ATP description file.
  - Starting from the second phase of the application, PTF starts querying the ATP server about details on application parameters, the latter reads the content of the ATP description file, queries the Omega Calculator and returns the list of valid values.
  - Once parameter details are acquired, PTF launches its exploration phase which is, at this point, similar to that of hardware and system parameters. The difference is that value assignment uses the ATP library instead of a PCP.
- **During RAT**
  - The application executable is launched along with other READEX components which includes all the components of READEX tool suite except PTF.
  - On receiving the enter region event from Score-P, RRL gets the optimal configuration for the ATPs from the application tuning model. Once the value assigment API call of the ATP library is encountered, the ATP library requests the RRL for the ATP value and RRL passes it to the ATP library. The value assignment is done by the ATP library.

6.5 API Functions

```
Parameters functions:

ATP_PARAM_DECLARE(const char *param_name, const char param_type, int
    default_value, const char *domain_name)
ATP_PARAM_ADD_VALUES(const char *param_name, void *vArray, int
    array_size, const char *domain_name)
ATP_PARAM_GET(const char *param_name, void *tp_address, const char
    *domain_name)

Constraints functions:

ATP_CONSTRAINT_DECLARE(const char *constraint_name, const char
    *constraint_expr, const char *domain_name)

Exploration functions:

ATP_EXPLORATION_DECLARE(const char *explorations_list, const char
    *domain_name)
```

More details on the library API are provided in Deliverable 4.5.

7 Summary

In this Deliverable we have given an overview of the implementation of the READEX tool suite from the parameters perspective. We described the integration of the parameter control plugins and the ATP plugin inside the READEX tool suite and gave an outline of the interaction between the different components.

A READEX Tuning Plugin has been implemented which supports tuning of hardware and software parameters. PCPs have been implemented for setting the values of CPU frequency, uncore frequency and energy performance bias. Two PCPs, one each for controlling the MPI and OpenMP parameters, are also part of the READEX tool suite

An ATP plugin has been developed that enable users to give hints on how to exploit dynamicity in the application. Through the ATP plugin, a user interface (ATP Library) is provided to specify different possible tunable application parameters and to get their values.

The implementation of PCPs and ATP plugin is finalized for the READEX tool suite. Future work will involve investigation of future parameters which will be presented in Deliverable 1.3.

References

[1] OpenMP Application Program Interface. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[2] Documentation about energy measurement infrastructure at Taurus phase 2. `https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/EnergyMeasurement`, Last accessed July 13, 2016.

[3] Andreas Gocht. D3.1: Final RRL architecture. Technical report, TUD, 2017.

[4] Andreas Gocht, Zakaria Bendifallah, Umbreen Sabir Mian, and Othman Bouizi. D1.1: Hardware and system-software tuning plugins. Technical report, TUD, Intel, 2016.

[5] Andreas Gocht, Umbreen Sabir Mian, Michael Lysaght, Venkatesh Kannan, Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy, Per Gunnar Kjeldsberg, Mohammed Sourouri, and Nico Reissmann. D4.2: Prototype READEX tool suite. Technical report, ICHEC, TUD, TUM, NTNU, IT4I, Intel, GNS, 2017.

[6] D. Hackenberg, T. Ilsche, J. Schuchart, R. Schöne, W.E. Nagel, M. Simon, and Y. Georgiou. HDEEM: High Definition Energy Efficiency Monitoring. In *Energy Efficient Supercomputing Workshop (E2SC)*, Nov 2014. DOI: 10.1109/E2SC.2014.13.

[7] Lubomir Riha, Tomas Brzobohaty, Alexandros Markopoulos, Ondrej Meca, and Tomas Kozubek. Massively Parallel Hybrid Total FETI (HTFETI) Solver. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '16, New York, NY, USA, 2016. ACM.

[8] Evan Rosser, Wayne Kelly, Bill Pugh, Dave Wonnacott, Tatiana Shpeisman, and Vadim Maslov. The Omega calculator. `https://github.com/davewathaverford/the-omega-project`.

[9] Evan Rosser, Wayne Kelly, Bill Pugh, Dave Wonnacott, Tatiana Shpeisman, and Vadim Maslov. The Omega project. `http://www.cs.umd.edu/projects/omega/`.

[10] Robert Schöne and Daniel Molka. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - R&D*, 29(3-4):231–239, 2014. DOI: 10.1007/s00450-013-0243-7.