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

GA no. 671657

READEX

Runtime Exploitation of Application Dynamism
for Energy-efficient eXascale computing

D4.3
READEX Tool Suite version 2

Document type:	Report
Dissemination level:	Public
Work package:	WP4
Editor:	Venkatesh Kannan (ICHEC-NUIG)
Contributing partners:	TUM, NTNU, TUD, ICHEC-NUIG, Intel
Reviewers:	IT4I, GNS, Intel
Version:	1.0

Document history

Version	Date	Author/Editor	Description
0.1	08-Jan-2018	Venkatesh Kannan (ICHEC-NUIG),	Structure
0.2	02-Feb-2018	Andreas Gocht, Umbreen Sabir Mian (TUD), Venkatesh Kannan (ICHEC-NUIG), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Nico Reissmann (NTNU) Uldis Locans (Intel)	for 1st review
0.3	16-Feb-2018	Andreas Gocht, Umbreen Sabir Mian (TUD), Venkatesh Kannan (ICHEC-NUIG), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Nico Reissmann (NTNU) Uldis Locans (Intel)	for 2nd review
1.0	28-Feb-2018	Andreas Gocht, Umbreen Sabir Mian (TUD), Venkatesh Kannan (ICHEC-NUIG), Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy (TUM), Per Gunnar Kjeldsberg, Nico Reissmann (NTNU) Uldis Locans (Intel)	for submission

Contents

1   Introduction

The READEX project develops a tools-aided approach to analyse and tune HPC applications for energy efficiency on Exascale systems. This is achieved using the READEX tool suite, the architecture and workflow of which were presented in Deliverable D4.1 [8]. In M18, the project delivered the alpha prototype of the READEX tool suite the features of which were summarised in Deliverable D4.2 [5]. The alpha prototype detected intra-phase and inter-phase dynamism in applications, and in response to intra-phase dynamism tuned three tuning parameters – processor core frequency using DVFS (dynamic voltage and frequency scaling), processor uncore frequency using UFS (uncore frequency scaling) and number of OpenMP threads – to determine and set optimal tuning configurations for runtime situations of significant regions.

The beta prototype of the READEX tool suite delivered in M30, which extends the alpha prototype, implements the features as described below:

1. **Design-Time Analysis (DTA)** As a first step in DTA, the tuning potential of an HPC application is quantified using two dynamism metrics: execution time and compute intensity. If an application is determined to have significant tuning potential, then experiments are performed during DTA to analyse the runtime situations (rts's) of significant regions in the application for intra-phase and inter-phase dynamism. These experiments are used to identify the optimal configurations for hardware, system software and application tuning parameters using multiple objectives. The results of the DTA experiments 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 into a *scenario*, the *selector* provides the best configuration determined for the tuning parameters during DTA for a given *scenario*.

   The major extensions to the beta prototype are the ability to tune application parameters using the *Domain-Knowledge Specification Interface* (DKSI), the support for input identifiers, and the extensions to the READEX plugin developed for evaluating the tuning success.

   DTA is performed by the Periscope Tuning Framework (PTF) in conjunction with Score-P that provides the instrumentation and measurement infrastructure. The ability to specify and tune application tuning parameters is provided through the newly developed ATP library which implements the DKSI. 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) implementing Runtime Application Tuning (RAT).

2. **Runtime Application Tuning (RAT)** Following the completion of DTA and the 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.

The scenario identification mechanism uses additional significant region identifiers, in the form of Score-P user parameters, that are included in the tuning model. Following this, the optimal configuration for the current scenario during the production run is selected from the tuning model and applied in a step called *switching*.

A major extension to the beta prototype is the extension to update the tuning model during RAT using a step called *calibration*, which uses machine learning to analyse runtime situtations that were unseen during DTA and adds them to the tuning model during RAT, and the implementation of advanced scenario classification and selection mechanisms.

In the beta prototype, switching is performed for the processor core frequency, uncore frequency, number of OpenMP threads and application parameters. 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.

2   Features of Beta Prototype

DTA identifies the optimal configurations of tuning parameters for runtime situations (rts's), i.e., individual executions of program regions (see Deliverable D4.1 [8], page 10). These optimal configurations are stored in a *tuning model* which is then passed to the RAT that automatically switches to the identified configurations dynamically during the application's production run.

2.1   Added Features of Design-Time Analysis (DTA)

DTA forms the first stage in the READEX tool suite. The beta prototype of the tool suite extends the alpha prototype by extending existing components and adding new components, thereby achieving the following goals that are summarised in Sections 2.1.1 through 2.1.4:

- Extension of DTA for Application Tuning Parameters (ATP).

- Extension for input identifiers.

- Advanced tuning model generation.

- Extension for evaluating the tuning success.

This section describes the new features that were added to the DTA stage in the beta prototype. Newly added components (colored red), and modified components (colored green) are illustrated in Figure 1.

2.1.1   Extensions for Application Tuning Parameters

The following components implement the support for ATPs, which is described in Deliverable D2.3:

ATP Library The ATP library implements the Domain-Knowledge Specification Interface (DKSI) defined in Deliverable D4.5. It allows to specify tuning parameters with their sets of values and their ATP domain. Variables in the same ATP domain can have constraints defining valid settings.

   During DTA, the ATP library generates the ATP Description File specifying ATP domains, ATPs, and constraints in the first application phase. In subsequent application phases, the ATP library receives the ATP settings configured by PTF through the online access interface from the RRL and assigns the values to application variables influencing the control flow.

ATP Server In contrast to the ATP library which is tied to the application itself, the ATP server is launched by PTF, and reads the contents of the ATP description file. The
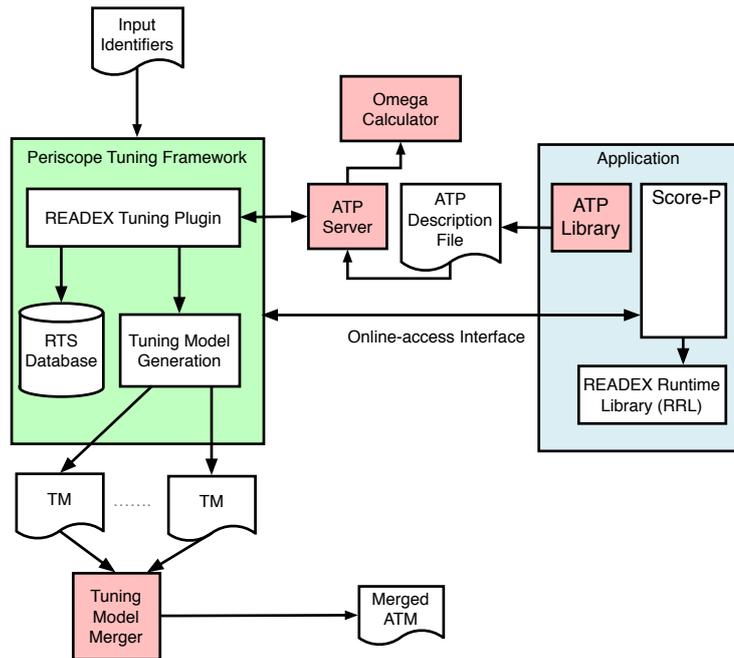
Figure 1: DTA support for input identifiers and application tuning parameters. Components developed since the alpha prototype are colored red, modified components green.

primary task of the ATP server is to respond to PTF's requests on ATPs. Some requests are simple, such as querying the list of ATP parameters, and the server can get the information by looking in the content of the ATP description file. Other requests are more complex, such as the generation of a list of valid points for the parameters, which requires the resolution of the constraints held between parameters. This requires the server to query a third party constraint solver software called the *Omega Calculator* [12].

Omega Calculator The Omega Calculator is an affine functions constraint solver, and is used to generate valid values for each ATP for the recorded constraints by filtering the tuning parameter values that do not satisfy the constraints. The Omega Calculator software is composed of the so-called *Omega Library*, which constitutes 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 [11]. One big advantage of using the Omega Calculator is the small computational time needed to solve the affine function based constraints, which makes it fit for use to solve the constraints at runtime.

READEX Tuning Plugin PTF performs intra-phase analysis by executing the READEX tuning plugin, which reads the objective(s) (Energy, CPU Energy, Execution Time or Total Cost of Ownership), the tuning parameters (CPU frequency, uncore frequency and the number of OpenMP threads), the search strategy (exhaustive, random, individual

or genetic) and the significant regions from the READEX configuration file. It also starts the ATP Server and retrieves the information about ATPs. PTF provides two new search strategies for ATPs: `exhaustive_atp` and `individual_atp`. The strategies retrieve the valid points for ATPs from the ATP Server. A new first tuning step was added to the READEX plugin that tunes the ATPs with the help of the new search strategies. At the end of the tuning step, the best configuration for the ATPs is fixed and the system-level tuning parameters are explored in the second tuning step as implemented in the alpha prototype.

2.1.2   Extensions for Input Identifiers

The following components implement the support for input identifiers as described in Deliverable D2.3:

RTS Database The information about each rts, including its call path, region identifiers, default objective values, and the best and the worst setting of the tuning parameters for the READEX tuning plugin is stored in the RTS database. The RTS database was extended for storing also the input identifiers that are read at the beginning of DTA from a file accompanying the input data set. In addition, the number of processes and threads is stored there as well and also handled as input identifiers.

READEX Tuning Plugin At the end of the READEX Tuning Plugin the input identifiers are passed to Tuning Model generation.

Tuning Model Generation Finally, a tuning model is generated from the knowledge stored in the RTS database. The tuning model generation was extended for supporting input identifiers. These are written into the TM and guide tuning model merging.

Tuning Model Merger For each DTA run with an input specification file, a separate tuning model is generated. These tuning models are merged by this new external tool into a single generic tuning model. The Tuning Model Merger then merges all the tuning models into one merged application tuning model file, which is then read at runtime to perform dynamic switching.

2.1.3   Advanced Tuning Model Generation

The following components implement the advanced clustering mechanisms as described in Deliverable D2.3:

Tuning Model Generation An advanced clustering mechanism was added to support clustering of rts system configurations for the generation of scenarios. The clusterer uses hierarchical clustering and proceeds in three phases: dendrogram generation, cluster generation, and scenario creation. Dendrogram generation builds a tree of the individual rts's to conveniently express the distance betweens these rts's in the tuning

parameter space. Cluster generation finds the optimal number of clusters based on the Calinski-Harabasz Index [3], and finally, scenario generation creates scenarios from the found clusters.

2.1.4   Evaluation of the Tuning Success

The following components were extended to provide feedback on the tuning success:

READEX Tuning Plugin The READEX Tuning Plugin was extended for computing the tuning success. In a very first tuning step, it determines the objective value for the default configuration of the machine and the application tuning parameters. This is the basis for the evaluation. Another added tuning step at the end of the tuning plugin determines the static best configuration for the phase and the best configurations for each significant region. It then configures the RRL with this determined best configuration and performs three experiments to evaluate the stability of the results. During the experiments the switching happens as during RAT. The experiments enable checking for the stability of the results. Finally, the tuning plugin outputs the theoretical achievements as the reduction due to static tuning and dynamic tuning. Details can be found in Deliverable D2.3.

2.2   Features of Runtime Application Tuning (RAT)

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

The beta prototype extends and adds to the modules that were already present in the alpha prototype. This version of RRL also includes the implementation of the *calibration* mechanism. Figure 2 shows the current design of the RRL. Modules that have been added or have undergone major changes are marked green. Red marks the modules that are developed as part of the READEX tool suite.

This section summarises the changes made to the RRL for the beta prototype since the alpha prototype, while details about the unchanged modules are available in the report about the alpha prototype in Deliverable D4.2 [5].

Control Center Being the central component of RRL, *Control Center* manages the communication of RRL with Score-P and also the communication among the components of the RRL. Information coming from Score-P is accumulated in the *Control Center*, from where this information is redirected to the respective components of RRL responsible for processing that particular information. The information coming from the Online
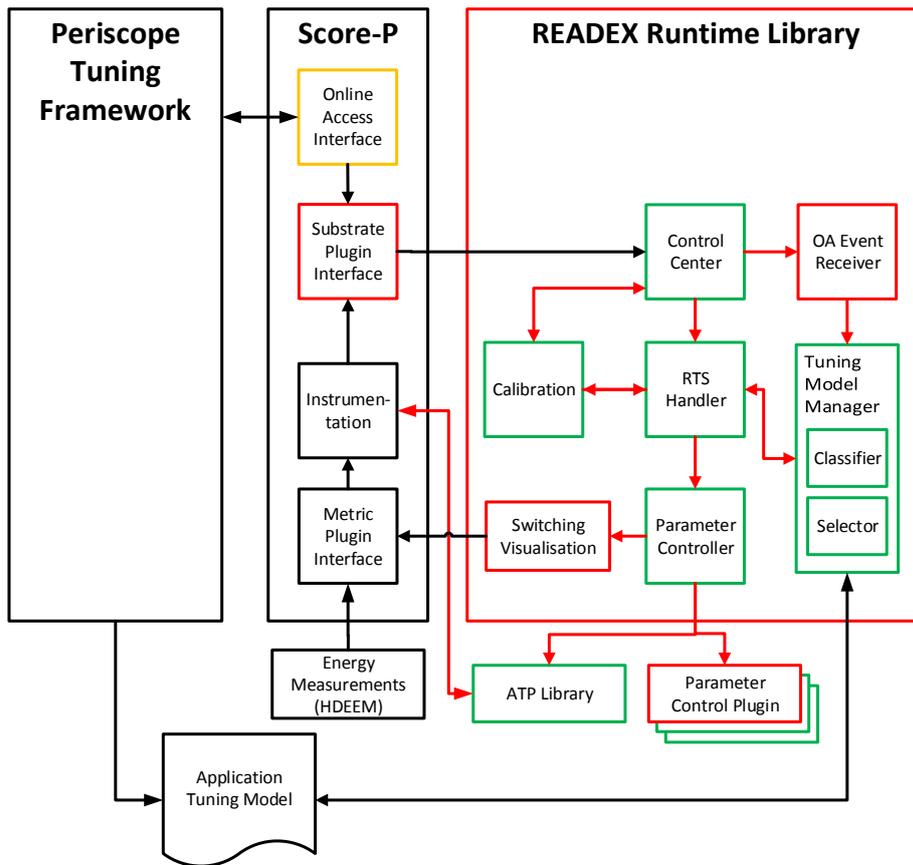
Figure 2: Current architecture of the READEX Runtime Library (RRL). Red: Implementation is part of the extensions made for the READEX tool suite. Green: Implementation has changed since the alpha prototype

Access interface of Score-P is forwarded to the *OA Event Receiver* and the information related to the runtime situations (rts's) is propagated to the *RTS Handler*. In the alpha prototype, the *Control Center* received the enter region, exit region and user parameter events for each rts which was then passed to the *RTS Handler*. In the beta prototype, the *Control Center* has been extended to receive the information for the thread creation and deletion in case of OpenMP applications. This information further goes down to the *RTS Handler*. The different enter region, exit region, and user parameter events are also now forwarded to the *Calibration* module.

RTS Handler The *RTS Handler* is extended to receive the thread creation and deletion information from the *Control Center* for OpenMP applications and passes it down to

the *Parameter Controller*. Furthermore, the *RTS Handler* is enhanced to accomodate the *Calibration* mechanism.

Upon receiving an enter region notification from Score-P via the *Control Center* during the application run, the *RTS Handler* checks with the Tuning Model Manager (TMM) whether the current region is a significant or an unknown region. If it is an unknown region, the calibration mechanism is called, and nothing else is done.

Tuning Model Manager (TMM) A *Tuning Model* is generated as a result of DTA and is required in RAT for running the application with the best configuration identified. The *Tuning Model Manager* holds the Tuning Model.

Advanced scenario classification and selection has been implemented in RAT TMM. During the application execution, the *RTS Handler* communicates with the TMM and passes the different rts's to the RAT TMM. The RAT TMM checks if the given rts is present in the Tuning Model. If yes, then 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*. Together with the *calibration* mechanism, the RAT TMM can now add new configurations to the TM. For details regarding the implementation of advanced scenario classification and configuration selection, please refer to the Deliverable D3.2 [6].

Parameter Controller As the name suggests, it controls the configuration switching of tuning parameters mentioned in Deliverable D1.1 [4]. The purpose of this component is to provide an interface between the RRL and the *Parameter Control Plugins* which are described below. This interface has been updated to pass the information for the creation and deletion of threads in case of OpenMP applications from the *RTS Handler* to the *Parameter Controller*.

Furthermore, the *Parameter Controller* is also extended to add the application tuning parameter (ATP) support. Two new subroutines for declaring a new ATP and to pass the configuration of the ATP stored in the Tuning Model to the application are added. The routine for declaring an ATP is invoked when an API call for `ATP_PARAM_DECLARE` from the DKSI is encounterd. Similarly the routine for getting the ATP optimal configuration is invoked when the program approaches the API call for `ATP_PARAM_GET` from the DKSI. The detailed syntax of the ATP related API calls from the DKSI is provided in the Deliverable D1.2 [9].

Parameter Control Plugins Switching of parameter settings is performed via the *Parameter Control Plugins*. For the beta prototype, the following parameter control plugins are available.

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

- OpenMP

A detailed description of all the above listed *Parameter Control Plugins* is available in Deliverable D1.1 [4].

Calibration The goal of the calibration is to handle unseen rts's during the production run of an HPC application. These unseen rts's may consist of already known regions which have some unknown parameters. Alternatively, these rts's may consist of completely new regions which have totally unknown characteristics.

A machine learning based approach is implemented which uses hardware performance events as feature vector for learning. Neural Networks (NN) is the algorithm of choice.

Feature vector (selected hardware performance events) for a given configuration of the processor core frequency normalized by the runtime of the region that results in the lowest energy consumption is given as input to the NN. The optimal processor core frequency for each region is the output of the NN.

For details regarding the concept and detailed implementation of the *Calibration* module, refer to the Deliverable D3.2 [6].

3   Using the Beta Prototype

3.1   User Guide

Associated with the beta prototype, we have created a detailed user guide with step-by-step instructions on using the tool suite on a given application. The detailed user guide is presented in Appendix A and it includes an example application to describe the steps to apply the tools, along with pointers to the example available on the Taurus cluster at TU Dresden which are accessible to the end-users.

3.2   Pathway

Pathway [10] is a high-level tool that supports an HPC user in managing application performance tuning with a formal workflow definition, workflow execution and workflow protocoling based on the standard Business Process Model and Notation (BPMN) format. By integrating all optimisation steps in a single workflow, Pathway allows structured and methodical performance engineering. Once defined, a Pathway workflow can then be executed by automatically triggering individual activities which may be execution of arbitrary tools or human tasks.

In the READEX project, Pathway is used to integrate the methodologies required to apply the tool suite and to simplify user interaction with the HPC system and the READEX tool suite. This was achieved by extending existing work items and creating new READEX-specific work items to achieve the steps required for the READEX workflow shown in Figure 3.

This workflow allows selecting the HPC system and the configuration (number of MPI processes and OpenMP threads) with which to execute the application in conjunction with the READEX tool suite. It is composed of tuning potential analysis using the `readex-dyn-detect` tool, design-time analysis using PTF and runtime application tuning using the RRL. The workflow allows the user to specify multiple inputs to the application for tuning potential and design-time analysis, and automatically prepares the READEX configuration file generated by `readex-dyn-detect` for use by PTF using inputs for the tuning parameters, objectives and other necessary inputs specified by the user in the Pathway application interface.

The results from different stages of the application's execution are summarised in the READEX browser that was newly created, an example of which is presented in Figure 4. The results that are summarised in the READEX browser include the list of applications that have been executed with the READEX tool suite, the summary of applying DTA each time to the application, dynamism information produced by the `readex-dyn-detect` tool obtained from the READEX configuration file, and the input used for the application to run the DTA experiments. Underneath the summary of results from DTA are three sections that provide details of the inter-phase and intra-phase dynamism that were identified and the tuning model file that was generated as a result of DTA.
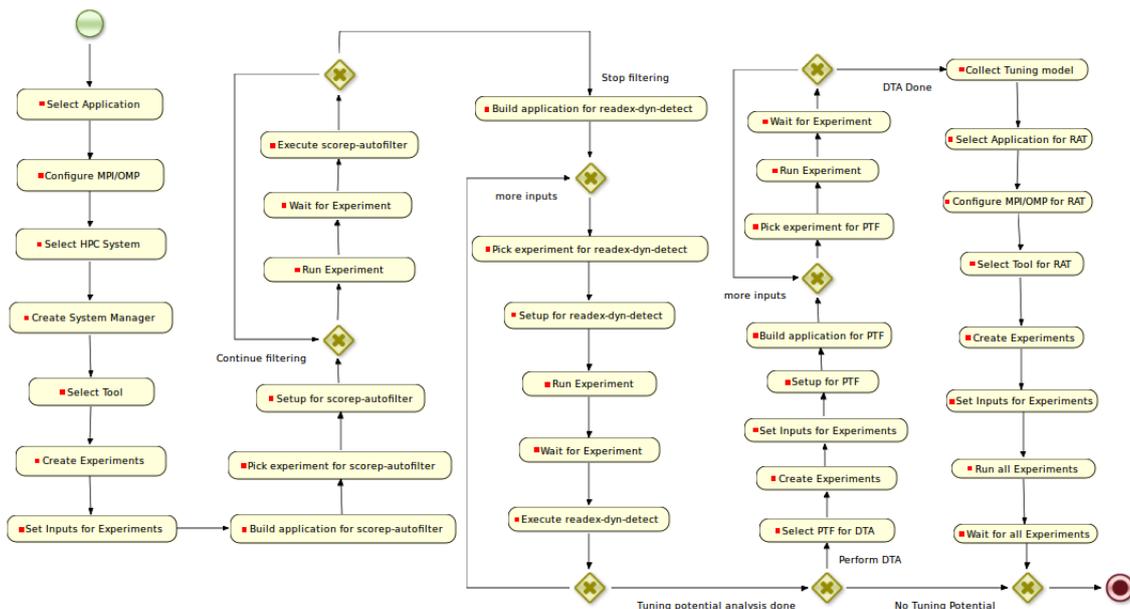
Figure 3: READEX workflow in Pathway

Figure 4: Pathway READEX browser sample

The READEX workflow can be loaded as a pre-existing workflow available in Pathway. A detailed documentation of the extensions to Pathway for the READEX project and a user guide will be produced after M30.

4   Summary

The features of the beta prototype of the READEX tool-suite summarised in this report cover the features that were targeted by M30 as outlined in the Deliverable D4.1 [8]. This prototype is currently being used to evaluate the benchmark applications selected for the READEX project as described in Deliverable D4.1 [8].

Continuing beyond M30, the READEX tool suite will be evaluated against the benchmark and production applications in the READEX test suite described in Deliverable D5.2. The beta prototype will also be made available to external users on the Taurus test-bed cluster at TU Dresden and for installation through RPM/DEB installation packages. Using the READEX tool-suite on a HPC cluster requires an energy measurement infrastructure such as HDEEM, RAPL or msr-safe.

References

[1] H. Brunst, D. Hackenberg, G. Juckeland, and H. Rohling. Comprehensive performance tracking with Vampir 7. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing*, pages 17–30, Berlin, 2010. Springer.

[2] Holger Brunst and Bernd Mohr. Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with VampirNG. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, May 2005.

[3] T. Calinski and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, 1974.

[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] Per Gunnar Kjeldsberg. D3.2: Final mechanisms for run-time detection, switching and calibration. Technical report, TUD, Intel, 2018.

[7] Per Gunnar Kjeldsberg, Michael Gerndt, Mohammed Sourouri, and Anamika Chowdhury. D2.1 analysis of tuning potential and scenario identification. Technical report, NTNU, TUM, 2016.

[8] Michael Lysaght, Kashif Iqbal, Joseph Schuchart, Andreas Gocht, Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy, Per Gunnar Kjeldsberg, Magnus Jahre, Mohammed Sourouri, David Horak, Lubomir Riha, Radim Sojka, Jakub Kruzik, Kai

Diethelm, and Othman Bouizi. D4.1: Concepts for the READEX tool suite. Technical report, ICHEC, TUD, TUM, NTNU, IT4I, Intel, gns, 2016.

[9] Umbreen Sabir Mian and Zakaria Bendifallah. D1.2 final tuning plugins. Technical report, TUD, Intel, 2017.

[10] Ventsislav Petkov, Michael Gerndt, and Michael Firbach. Pathway: Performance analysis and tuning using workflows. In *Proceedings of the IEEE 10th International Conference HPCC_EUC 2013*, pages 792–799, Nov 2013.

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

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

[13] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, 2015.

A   User Guide

This section describes how to use the READEX tool suite according a simple workflow:

1. Instrument the application with Score-P. (Section A.2)

2. Perform design-time analysis of application to create tuning model. (Section A.3)

3. Use the tuning model during the production run of the application for runtime tuning. (Section A.4)

A.1   Modules on Taurus

The tools in the READEX tool suite are accessible through modules created either by the continuous integration process or the beta release of the tool suite. Users in the `p_readex` group may use either, while those in `p_readextest` can only use the beta release.

Depending on the choice of compilers used for the application (GCC or Intel), load one of these modules to use the READEX tools that are required to analyse and tune an application at the different steps in the workflow from Section 1.

A.1.1   Continuous integration

Load the continuous integration modules on Taurus as follows:

- For `gcc/6.3.0` and `bullxmpi/1.2.8.4`:

```
module use /projects/p_readex/modules
module load readex/ci_readex_bullxmpi1.2.8.4_gcc6.3.0
```

- For `intel/2017.2.174` and `intelmpi/2017.2.174`:

```
module use /projects/p_readex/modules
module load readex/ci_readex_intelmpi2017.2.174_intel2017.2.174
```

A.1.2   Beta release

Load the beta release modules on Taurus as follows:

- For `gcc/6.3.0` and `bullxmpi/1.2.8.4`:

```
module load readex/beta_gcc6.3.0
```

- For `intel/2017.2.174` and `intelmpi/2017.2.174`:

```
module load readex/beta_intel2017.2.174_intelmpi2017.2.174
```

A.2 Application instrumentation

A.2.1 Build application with Score-P

The READEX tool suite is based on instrumenting an application with Score-P. Instrumentation inserts measurement probes into the source code of the application. This can be done by the compiler, other software tools, or manually. Detailed documentation on Score-P and the instrumentation features can be found at `www.score-p.org`.

1. Modify the application's makefile for instrumentation with Score-P. Prepend the compilation with the `scorep` command. For example,

   Replace `MPICXX = mpic++ -fopenmp`
   by `MPICXX = scorep -mpp=mpi mpic++ -fopenmp`

   The `scorep` command switches on compiler instrumentation of program functions as well as instrumentation of MPI routines and OpenMP regions.

   Use `-mpp=mpi` for MPI applications and `-mpp=none` for non-MPI applications.

2. Build the application. Note that Score-P and the application have to be built with the same compiler.

3. Run the application as like the uninstrumented version.

Outcome: Compiler instrumentation of the application is performed; upon application execution, Score-P creates a profile (`profile.cubex`) file in the `scorep-<xyz>` directory at the execution location.

A.2.2 Filtering

The probes inserted in the application through instrumentation add overhead to the application execution and thus can make any measurements and tuning efforts wasted time. Therefore, it is essential to make sure that the instrumentation overhead is below a certain limit. Therefore, this section focuses on giving you advice on the support in Score-P for reducing the measurement overheads. To measure the overhead, first measure the execution without instrumentation and then measure it with instrumentation.

To reduce the overhead from instrumentation to an acceptable level,

1. First try to reduce the overhead with runtime and compile time filtering as described in Sections A.5.1 and A.5.2, respectively.

2. You may also remove MPI and OpenMP region instrumentation overhead as described in Section A.5.3.

3. Then switch on the energy measurements with HDEEM since it has a much higher overhead than just time measurements as described in Section A.5.4. Verify the overhead again.

   As an alternative, RAPL can be used for energy measurement, which has lesser overhead than HDEEM. Note that the energy measurements from RAPL may not be precise enough. For instance, a reading time less than 40 ms (that is 40 ms function execution time and 1 ms sampling rate) may result in approximately 2.5% error.

4. If the overhead is still too high, consider manual instrumentation of those regions that are relevant for the READEX tool suite as described in Section A.5.5.

Do not proceed to energy tuning if the overhead is too high.

A.2.3   Phase region instrumentation

Specify the 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
```

A phase region is a repetitive, single-entry and exit region, typically the body of the main progress loop of the application. If the phase region is not known beforehand, it may be useful to look at the `profile.cubex` file generated after running the `scorep-autofilter` tool with a performance analysis tool like CUBE [13].

Example   The `for-loop` body in `Integrate::run()` is annotated as a phase region as shown in the example in Section A.7.2.

A.2.4   Application tuning parameter instrumentation

Specify the application tuning parameters: It is also possible to optionally exploit application level tuning using the READEX tool suite. This requires some additional manual code annotation and instrumentation to pinpoint the parts of the code that can be exploited as application tuning parameters and annotate them with certain API functions.

This is enabled in READEX using the ATP (Application Tuning Parameter) library and the procedure for this is described in Section A.6.1.

A.3   Design-time Analysis (DTA)

A.3.1   Tuning Potential Analysis

The first step in the DTA is to detect and analyze the dynamism of the application using `readex-dyn-detect`. The tool automatically identifies the significant regions that are subject to the READEX tuning methodology and generates a report on the potentially exploitable dynamism in these regions.

The `readex-dyn-detect` tool requires a single phase region, which is to be instrumented as described earlier in Section A.2.3.

Perform the following steps to use `readex-dyn-detect`:

1. Build the application with `scorep --online-access --user --thread=none` for the manually annotated phase region and add `--nocompiler` if the application is manually instrumented.

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_sec>
                     -p <phase_region_name>
                     -c <compute_intensity_variation_threshold>
                     -v <execution_time_variation_threshold_in_percent>
                     -w <region_execution_time_weight_wrt_phase_execution_time_in_percent>
                     -r <Configuration file name without extension>
                     -f <RADAR_report_file_name>
                     <path_to_cubex_file>/profile.cubex
   ```

   The command line options have the following meaning:

   -t This threshold specifies the minimal mean execution time of regions that are to be considered as significant regions. Use a value larger than 0.1 (100 ms).

   -p Name of the phase region as given in the instrumentation.

   -c This is the required minimal standard deviation of the compute intensities of significant regions with a weight above the given threshold, such that intra-phase dynamism due to compute intensity variation is reported.

-v This is the required minimal standard deviation of the execution time of instances of significant regions in percent of the mean region's execution time, such that intra-phase dynamism is reported. It is also used to decide whether inter-phase dynamism exists. Only if the standard variation of the phase time in percent of the mean phase time is greater, inter-phase dynamism is reported.

-w This threshold specifies the minimal weight of a region such that any dynamism due to time variation or compute intensity variation is reported.

-r This is the desired name for the READEX configuration file to be created by `readex-dyn-detect` without the file name extension.

-f If a file name is given, the report is generated in LATEX form to include it into the RADAR report.

4. The results of `readex-dyn-detect` are summarized 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`.

   Alternatively, the `readex_config.xml` file may be manually created from this template and used as input for PTF without applying `readex-dyn-detect` if the significant regions are already known.

   Note: `readex-dyn-detect` currently ignores MPI and shared memory regions in the significant regions analysis.

Outcome: The `readex_config.xml` file containing the tuning potential summary, the list of significant regions, and the intra-phase and inter-phase dynamism due to variation in the execution time and compute intensity.

Section A.7.5 presents an example.

A.3.2   Specify Criteria for DTA

The next step of the DTA is to update the `readex_config.xml` file generated by the `readex-dyn-detect` tool with additional criteria for the design-time analysis experiments performed by the Periscope Tuning Framework (PTF). The steps to update the `readex_config.xml` file are as follows:

1. Specify the tuning parameters: The READEX tuning plugin supports three tuning parameters – processor core frequency, uncore frequency and the number of OpenMP threads. A minimum of one tuning parameter must be specified. Specify the ranges (minimum, maximum and the step size) for the processor core frequency in kHz and for the uncore frequency in 100 million Hz. For OpenMP threads, specify the lower bound and the step size to increment to the next value.

Example

```
<tuningParameter>
  <frequency>
    <min_freq>1200000</min_freq>
    <max_freq>2400000</max_freq>
    <freq_step>500000</freq_step>
  </frequency>
  <uncore>
    <min_freq>10</min_freq>
    <max_freq>30</max_freq>
    <freq_step>2</freq_step>
  </uncore>
  <openMPThreads>
    <lower_value>1</lower_value>
    <step>2</step>
  </openMPThreads>
</tuningParameter>
```

2. Specify the objectives: Specify at least one objective from Energy, Execution Time, CPU Energy, Energy Delay Product, Energy Delay Product Squared, CPUEnergy, Total Cost of Ownership (TCO). The normalized version of each of the objectives can also be specified. The plugin measures the objective values for all the specified objectives, but tunes the application only for the objective that is specified first.

Example

```
<objectives>
  <objective>Energy</objective>
  <objective>NormalizedEnergy</objective>
  <objective>Time</objective>
  <objective>NormalizedTime</objective>
  <objective>EDP</objective>
  <objective>NormalizedEDP</objective>
  <objective>ED2P</objective>
  <objective>NormalizedED2P</objective>
  <objective>CPUEnergy</objective>
  <objective>NormalizedCPUEnergy</objective>
  <objective>TCO</objective>
  <objective>NormalizedTCO</objective>
</objectives>
```

To compute TCO, the CostPerJoule and CostPerCoreHour also needs to be specified.

```
<Configuration>
  <CostPerJoule>0.00000008</CostPerJoule>
  <CostPerCoreHour>1.0</CostPerCoreHour>
</Configuration>
```

3. Specify the energy metrics: Specify the energy plugin name and associated metric names. For `hdeem_sync_plugin`, it's possible to measure the energy for the whole node and for the CPUs. The energy metrics should be specified under `<periscope> </periscope>`.

Example

```
<periscope>
  <metricPlugin>
    <name>hdeem_sync_plugin</name>
  </metricPlugin>
  <metrics>
    <node_energy>hdeem/BLADE/E</node_energy>
    <cpu0_energy>hdeem/CPU0/E</cpu0_energy>
    <cpu1_energy>hdeem/CPU1/E</cpu1_energy>
  </metrics>
</periscope>
```

To specify the RAPL counter energy plugin **x86_energy_sync_plugin**, use the configuration as follows:

Example

```
<periscope>
  <metricPlugin>
    <name>x86_energy_sync_plugin</name>
  </metricPlugin>
  <metrics>
    <node_energy>x86_energy/BLADE/E</node_energy>
    <cpu0_energy>x86_energy/CORE0/E</cpu0_energy>
    <cpu1_energy>x86_energy/CORE1/E</cpu1_energy>
  </metrics>
</periscope>
```

4. Specify a search algorithm: Specify a single search algorithm from exhaustive, random, individual or genetic search. For the random search strategy, specify the number of system configurations that the plugin should explore. For the individual search, specify the number of tuning parameter values to *keep* in the search space. Individual search tunes the parameters independently. The number of best settings (keep factor) are taken into account when tuning the next parameter. For the genetic search, specify the population size, the maximum number of generations and the timer to set an upper limit on the tuning execution time.

Example

```
<periscope>
  <searchAlgorithm>
    <name>exhaustive</name>
    <name>random</name>
      <samples>2</samples>
    <name>individual</name>
      <keep>2</keep>
    <name>gde3</name>
      <populationSize>10</populationSize>
      <maxGenerations>10</maxGenerations>
      <timer>20</timer>
  </searchAlgorithm>
</periscope>
```

If Application Tuning Parameters are used, you also specify the search strategy for ATP tuning (see Section A.6.2).

5. Specify the tuning model file name: The generated tuning model file name can also be specified under `<periscope> </periscope>`

Example

```
<periscope>
  <tuningModel>
    <file_path>./tuning_model.json</file_path>
  </tuningModel>
</periscope>
```

Optionally, if the Application Tuning Parameter (ATP) library is used, then the details for the ATP library should be included in the READEX configuration file as outlined in Section A.6.2.

A.3.3   Tuning Model Creation

After updating the `readex_config.xml` file for use by PTF, use the following steps to perform design-time analysis using PTF as explained using a slurm job script for the miniMD application as an example.

1. Build the application with instrumentation as discussed in Section A.2.3 (`scorep --online-access --user`) for the instrumented phase region. Additionally, you may optionally use the Score-P options that are required to specify compile-time filtering, MPP and thread instrumentation options. Refer to the Score-P documentation for this.

2. Set the number of nodes to at least 2 (line 4), and allocate enough memory per CPU to fit the application as shown in line 9. In general, if `N > 1` nodes are allocated for this job, then PTF will use one node for the tool's agents and the remaining `N-1` nodes for the application processes.

3. Use the parameter control plugins compatible with Score-P and PTF as shown in line 27, and set the environment variable with the tuning parameters as shown in line 28.

4. Load the `scorep-hdeem sync` plugin for energy measurements compatible with the Score-P built for the READEX toolsuite, and set the environment variables as shown in lines 31–38.

5. Apply PTF on the application with the `psc_frontend` command as shown in lines 40–48. Specify the instrumented phase region name for the option `--phase`, the readex_tuning plugin for `--tune` and the readex configuration file for `--config-file`.

   The options `--info` and `--selective-info` are only used for debug messages, and are not mandatory. For more debug output, set the `--info=<max_info_level>` between 2

and 7, and `--selective-info=<comma␣separated␣list␣of␣information␣levels>`. For more information about other options, see `psc␣frontend --help`.

This will produce a tuning model in the execution directory under the name specified in the `readex␣config.xml` file, or `tuning␣model.json` if unspecified.

```
1    #!/bin/sh
2
3    #SBATCH --time=5:00:00   # walltime
4    #SBATCH --nodes=2  # number of nodes requested; 1 for PTF and remaining for application run
5    #SBATCH --tasks-per-node=8 # number of processes per node for application run
6    #SBATCH --cpus-per-task=1
7    #SBATCH --exclusive
8    #SBATCH --partition=haswell
9    #SBATCH --mem-per-cpu=2500M   # memory per CPU core
10   #SBATCH -J "miniMD_PTF"   # job name
11   #SBATCH -A p_readex
12
13   echo "run PTF begin."
14
15   NP=8 # check against --ntasks and tasks-per-node
16
17   module purge
18   module use /projects/p_readex/modules
19   #module load readex/beta_gcc6.3
20   module load readex/ci_readex_bullxmpi1.2.8.4_gcc6.3.0
21
22   INPUT_FILE=in3.data #in.lj.miniMD
23   PHASE=INTEGRATE_RUN_LOOP
24
25   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
26
27   export SCOREP_SUBSTRATE_PLUGINS=rrl
28   export SCOREP_RRL_PLUGINS=cpu_freq_plugin,uncore_freq_plugin
29   export SCOREP_RRL_VERBOSE="WARN"
30
31   module load scorep-hdeem/sync-xmpi-gcc6.3
32   export SCOREP_METRIC_PLUGINS=hdeem_sync_plugin
33   export SCOREP_METRIC_PLUGINS_SEP=";"
34   export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_CONNECTION="INBAND"
35   export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_VERBOSE="WARN"
36   export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_STATS_TIMEOUT_MS=1000
37
38   export SCOREP_MPI_ENABLE_GROUPS=ENV
39
40   psc_frontend --apprun="./miniMD_openmpi_ptf -i $INPUT_FILE"
41                --mpinumprocs=$NP
42                --ompnumthreads=1
43                --phase=$PHASE
44                --tune=readex_tuning
45                --config-file=readex_config.xml
46                --force-localhost
47                --info=7
48                --selective-info=AutotuneAll,AutotunePlugins
49
50   echo "run PTF done."
```

To use the RAPL counter energy plugin change from lines 31–36 with the following:

```
1   module load scorep_plugin_x86_energy
2   export SCOREP_METRIC_PLUGINS=x86_energy_sync_plugin
3   export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN=*/E
4   export SCOREP_METRIC_PLUGINS_SEP=";"
5   export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN_CONNECTION="INBAND"
6   export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN_VERBOSE="WARN"
7   export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN_STATS_TIMEOUT_MS=1000
```

A batch job script to apply PTF for design-time analysis and create a tuning model for the miniMD application is available in

```
/projects/p_readextest/miniMD/run_ptf.sh
```

and is submitted as

```
sbatch run_ptf.sh
```

For different applications, `run_ptf.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.

Outcome:

  - A printed summary of the created scenarios, the properties found in each scenario, the optimum and the worst scenarios for the phase, the measured objective values for the phase in each scenario, the best configuration for each rts, the static and dynamic energy savings for the rts's, and the static energy savings for the whole phase.

  - A `tuning_model.json` file containing the list of rts's that were tuned by the plugin, the scenarios into which they are classified, and the best configuration for each scenario.

A.4   Runtime Application Tuning (RAT)

A.4.1   Production Run with Tuning Model

The following steps describe how to use RRL to tune the application during its production run and compare the execution time and energy consumption with an untuned run of the application.

1. If Application Tuning Parameters are exploited in the application then the ATP related instrumentation functions should remain in the code.

2. Use an uninstrumented verion of the application to compare its energy consumption and execution time against the version tuned with RRL.

3. For the application run tuned with RRL, use the application built for analysis with PTF as described in Section A.3.

4. Set the number of nodes to run the application on (line 4), and allocate enough memory per CPU to fit the application (line 10). Here, the number of nodes required is the same as the number of nodes on which to run the application.

5. For the untuned run of the application (lines 28–66) perform the following steps:

   (a) Disable Score-P profiling and tracing (lines 29 and 30), and set the Score-P substrate plugins, RRL tuning plugins and the tuning model to empty (lines 31–33).

   (b) Before running the uninstrumented version of the application (line 41), start the HDEEM energy measurements on all nodes (line 37–38) and get the start timestamp (line 39).

   (c) After the application run is complete, stop the HDEEM measurements and print the statistics from all nodes into a file `hdeem.out` (lines 47–49), and get the end timestamp (line 43).

   (d) Aggregate the energy consumption for the untuned run of the application from `hdeem.out` (lines 47–60).

6. For the RRL-tuned run of the application (lines 68–106) perform the following steps:

   (a) Disable Score-P profiling and tracing (lines 69 and 70), set the Score-P substrate plugins to `rrl`, RRL plugins to the tuning plugins to use (`cpu_freq_plugin` and `uncore_freq_plugin` in this example) and the tuning model to the file generated by PTF (lines 71–73).

   (b) Before running the RRL-tuned version of the application (line 81), start the HDEEM energy measurements on all nodes (line 77–78) and get the start timestamp (line 79).

   (c) After the application run is complete, stop the HDEEM measurements and print the statistics from all nodes into a file `hdeem.out` (lines 87–89), and get the end timestamp (line 83).

(d) Aggregate the energy consumption for the RRL-tuned run of the application from `hdeem.out` (lines 91–105).

```
1   #!/bin/sh
2
3   #SBATCH --time=2:00:00
4   #SBATCH --nodes=1
5   #SBATCH --ntasks=8
6   #SBATCH --tasks-per-node=8
7   #SBATCH --cpus-per-task=1
8   #SBATCH --exclusive
9   #SBATCH --partition=haswell
10  #SBATCH --mem-per-cpu=2500M
11  #SBATCH -J "miniMD_rrl"
12  #SBATCH -A p_readex
13
14  module use /projects/p_readex/modules
15  module load readex/ci_readex_bullxmpi1.2.8.4_gcc6.3.0
16
17  energy_label="Energy"
18  rm -rf host_names.out
19  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 hostname >> host_names.out
20
21  #####
22  # application-specific setup here
23  INPUT_FILE=in3.data #in.lj.miniMD
24  #####
25
26  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
27
28  # start plain run
29  export SCOREP_ENABLE_PROFILING="false"
30  export SCOREP_ENABLE_TRACING="false"
31  export SCOREP_SUBSTRATE_PLUGINS=""
32  export SCOREP_RRL_PLUGINS=""
33  export SCOREP_RRL_TMM_PATH=""
34  export SCOREP_MPI_ENABLE_GROUPS=ENV
35
36  # start measurements
37  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 clearHdeem
38  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 startHdeem
39  start_time=$(($(date +%s%N)/1000000))
40  # run untuned application
41  srun ./miniMD_openmpi_plain -i $INPUT_FILE
42  # stop measurements
43  stop_time=$(($(date +%s%N)/1000000))
44  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 stopHdeem
45  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 sleep 5
46  exec < host_names.out
47  while read host_name; do
48    srun -N 1 -n 1 --ntasks-per-node=1 -c 1 --nodelist=$host_name checkHdeem >> hdeem.out
49  done
50
51  # aggregate energy measurements from HDEEM
52  energy_total=0
53  if [ -e hdeem.out ]; then
54    exec < hdeem.out
55    while read max max_unit min min_unit average average_unit energy energy_unit; do
56      if [ "$energy" == "$energy_label" ]; then
57        read blade max_val min_val average_val energy_val
58        energy_total=$(echo "${energy_total} + ${energy_val}" | bc)
```

```
59        fi
60     done
61     time_total=$(echo "${stop_time} - ${start_time}" | bc)
62     echo ""
63     echo "Untuned run: Total time = $time_total ms, Total energy = $energy_total J"
64     rm -rf hdeem.out
65  fi
66  # end plain run
67
68  # start RRL-tuned run
69  export SCOREP_ENABLE_PROFILING="false"
70  export SCOREP_ENABLE_TRACING="false"
71  export SCOREP_SUBSTRATE_PLUGINS="rrl"
72  export SCOREP_RRL_PLUGINS="cpu_freq_plugin,uncore_freq_plugin"
73  export SCOREP_RRL_TMM_PATH="tuning_model.json"
74  export SCOREP_MPI_ENABLE_GROUPS=ENV
75
76  # start measurements
77  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 clearHdeem
78  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 startHdeem
79  start_time=$(($(date +%s%N)/1000000))
80  # run RRL-tuned application
81  srun ./miniMD_openmpi_ptf -i $INPUT_FILE
82  # stop measurmenents
83  stop_time=$(($(date +%s%N)/1000000))
84  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 stopHdeem
85  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 sleep 5
86  exec < host_names.out
87  while read host_name; do
88     srun -N 1 -n 1 --ntasks-per-node=1 -c 1 --nodelist=$host_name checkHdeem >> hdeem.out
89  done
90
91  # aggregate energy measurements from HDEEM
92  energy_total=0
93  if [ -e hdeem.out ]; then
94     exec < hdeem.out
95     while read max max_unit min min_unit average average_unit energy energy_unit; do
96        if [ "$energy" == "$energy_label" ]; then
97           read blade max_val min_val average_val energy_val
98           energy_total=$(echo "${energy_total} + ${energy_val}" | bc)
99        fi
100    done
101    time_total=$(echo "${stop_time} - ${start_time}" | bc)
102    echo ""
103    echo "RRL-tuned run: Total time = $time_total ms, Total energy = $energy_total J"
104    rm -rf hdeem.out
105 fi
106 # end RRL-tuned run
```

This batch job script is available in

```
/projects/p_readextest/miniMD/run_rrl.sh
```

and is submitted as

```
sbatch run_rrl.sh
```

For different applications, `run_rrl.sh` can be reused by updating the command to run the application in lines 41 and 81. This script is to be run from the location with the application's executable.

Outcome:

- The total execution time and energy consumption of the untuned run of the application and the run tuned by RRL are printed for comparison.

A.4.2 Visualise Configuration Switching

There are two ways of visualising the configuration switching:

1. A visualization plugin that shows the RRL perspective to the switching, i.e. the configuration that is supposed to be applied. It can be used during DTA and RAT.

2. The Score-P asynchronous plugins that show what actually happens in the processor. They can just be applied during RAT.

Using the visualisation plugin Since visualization is implemented as a synchronous plugin, Score-P supports this only in profiling mode, so to get the metrics in trace, tracing has to be set.

```
export SCOREP_ENABLE_TRACING=true
```

1. Set the environment variables to specify the metric plugin from RRL for visualization of tuning parameters as metrics in Vampir.

```
export SCOREP_METRIC_PLUGINS="scorep_substrate_rrl"
```

2. Set the environment variable to specify the tuning parameters which need to be added to trace. For the hardware and software tuning parameters, names of the PCPs are used. All of the hardware and software parameters can be loaded by simply setting the environment variable to "*". This will add all the loaded tuning parameters into the Vampir [1, 2] trace and the user can visualize the switching of values of each tuning parameter for each region of the application in the trace. Application Tuning Parameters (ATP) need to be explicitly specified. To load ATPs, the value should be set equal to 'ATP/<atp_name>' where atp_name is the name of the ATP as specified by the user. The prefix 'ATP/' is required to recognize the ATPs.

```
export SCOREP_METRIC_SCOREP_SUBSTRATE_RRL="ATP/<atp_name>, <pcp_name>"
```

For example, the environment variables to specify the RRL as metric plugin and view the processor core frequency switching in trace in Vampir can be set as follows:

```
export SCOREP_METRIC_PLUGINS="scorep_substrate_rrl"
export SCOREP_METRIC_SCOREP_SUBSTRATE_RRL="cpu_freq_plugin"
```
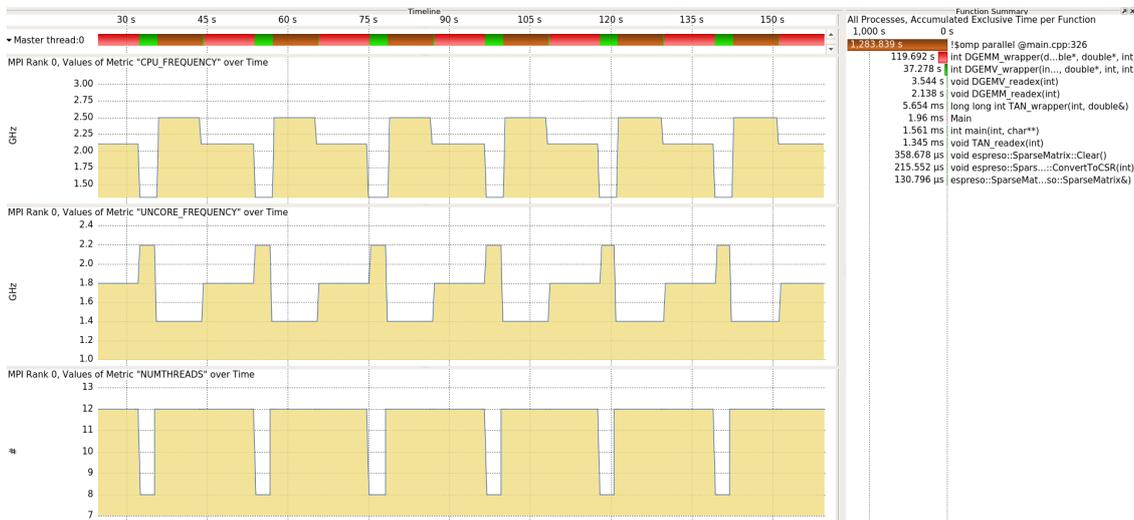

Figure 5: Vampir trace showing the switching of CPU_FREQUENCY, UN-CORE_FREQUENCY and NUMTHREADS for Blasbench benchmark

An example trace showing the switching of different configurations during RAT is given in Figure 5. The Score-P tracing is enabled and the visualization plugin is applied during the RAT phase for Blasbench benchmark which traces all the tuning parameters specified through parameter control plugins. The tuning parameters in Figure 5 are named as CPU_FREQUENCY, UNCORE_FREQUENCY and NUMTHREADS. The visualization plugin shows the configurations which have been set through RRL. To confirm that these configurations are actually set in the processor, Score-P asynchronous plugins, which are explained next, can be used.

Using the asynchronous Score-P sampling plugins   To use the asynchronous PAPI and uncore plugin, and to visualize the processor core and uncore frequencies, please add the following lines to your script:

```
module load scorep-uncore
module load scorep-apapi

export SCOREP_ENABLE_TRACING=true
export SCOREP_ENABLE_PROFILING=false
export SCOREP_METRIC_PLUGINS="apapi_plugin,upe_plugin"
export SCOREP_METRIC_APAPI_PLUGIN="PAPI_TOT_CYC"
export SCOREP_METRIC_APAPI_INTERVAL_US=10000
export SCOREP_METRIC_UPE_PLUGIN="hswep_unc_cbo0::UNC_C_CLOCKTICKS"
```

```
export UPE_INTERVAL_US=10000
export SCOREP_EXPERIMENT_DIRECTORY=<location_for_trace_file>
```

The trace file generated will be placed in the folder specified by SCOREP_EXPERIMENT_DIRECTORY. This can be viewed using Vampir.

Figure 6 shows the trace for the APAPI_TOT_CYC and UNC_C_CLOCKTICKS traced using the asynchronous PAPI and uncore plugins respectively. Both the traces presented in Figure 5 and Figure 6 are obtained in the same RAT run of Blasbench benchmark. The APAPI_TOT_CYC trace in Figure 6 confirms the trace of CPU_FREQUENCY in Figure 5. The APAPI_TOT_CYC trace in Figure 6 shows that for region "!$omp parallel@main.cpp" the frequency is first set to 2.5GHz according to the CPU_FREQUECY set by RRL but then goes down to zero whereas the CPU_FREQUENCY shown in Figure 5 stays at 2.5GHz for the entire duration of "!$omp parallel@main.cpp" region. The reason for this difference is that this is an omp parallel region and the Master thread goes to sleep while waiting for other threads to finish. The UNC_C_CLOCKTICKS trace in Figure 6 also confirms that the value of UNCORE_FREQUENCY is set as instructed by RRL.

Figure 6: Vampir trace showing the PAPI_TOT_CYC and UNC_C_CLOCKTICKS recorded using the Score-P asynchronous PAPI and uncore plugin respectively

Details about the plugins can be found at: `https://github.com/score-p/scorep_plugin_apapi` and `https://github.com/score-p/scorep_plugin_uncore`.

A.5   Filtering and Manual Instrumentation

A.5.1   Runtime Filtering

The first way to reduce the instrumentation overhead is to suppress the measurements done
by Score-P for instrumented regions. This is called runtime filtering of regions. READEX
provides the `scorep-autofilter` tool that inspects a generated profile and creates a filter
file for guiding runtime filtering. This file includes the names of too fine-granular regions that
are dominated by the measurement overhead.

1. Apply the `scorep-autofilter` tool on the `profile.cubex` file as follows:

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

   Choose a value to use as a threshold, for example 100 ms (`-t 0.1`) for regions to be
   considered for the significant region analysis. This will create a filter file with `.filt`
   extension. The user of the tool-suite can decide the value of the threshold depending
   on the amount of instrumenation overhead that they wish to retain for the analysis of
   regions in the application. The higher the threshold value, the lower will be the number
   of significant regions and the resulting instrumentation overhead.

2. It is advisable but not required to rerun the application and `scorep-autofilter` to detect
   additional fine granular regions that were missed in the previous step because their exe-
   cution time was increased by the measurement overhead of nested regions. This requires
   that the environment variable `SCOREP_FILTERING_FILE` is to be set to the filter file name
   (including the `.filt` extension) before rerunning the application.

   Apply `scorep-autofilter` to the new profile. Be careful not to overwrite the current
   filter file. Copy the newly found region names into the original filter file.

   Repeat this step until no more regions were found.

Outcome: A filter file with `.filt` extension containing the application regions that Score-P
will not measure.

Section A.7.1 presents an example.

A.5.2   Compile-time Filtering

Runtime filtering only suppresses the measurements while the overhead for the probes is
still there. You can apply the filter file also during instrumentation of the application to
suppress the insertion of probes for the given regions. Please check the Score-P user manual
for details on how to perform compile-time filtering. It is advisable that the user do this
whenever possible since each existing instrumentation interrupts the program flow during its
execution.

A.5.3   Filtering OpenMP and MPI regions

You can remove instrumentation of MPI routines and OpenMP regions as follows:

- **Filtering OpenMP regions:** To skip the instrumentation of OpenMP regions, the option `--thread=none` should be used. As a side-effect, no instrumented regions should occur inside of parallel regions. Otherwise, a runtime error will occur. Instead of switching off instrumentation of all OpenMP regions, you can also disable regions selectively via

  ```
  --opari="--disable=omp:single,master,atomic,critical,barrier"
  ```

  This will instrument parallel regions and nested instrumented regions would be handled as expected by Score-P.

- **Filtering MPI regions:** To disable measurements for MPI routines, you can add the following line to your batch script:

  ```
  export SCOREP_MPI_ENABLE_GROUPS=ENV
  ```

  It suppresses instrumentation for all MPI routines except `MPI_Init`, `MPI_Finalize` and other environment routines. These are required during DTA with the Periscope Tuning Framework.

A.5.4   Energy Measurements

Due to the overhead of energy measurements on Taurus with hdeem for application profiling with Score-P of about 5 ms, it is necessary to check the overhead when the energy measurements are switched on.

For energy measurements, load the hdeem module compatible with the compiler that was used to build the READEX tool suite.

```
module load scorep-hdeem/sync-xmpi-gcc6.3
(or)
module load scorep-hdeem/sync-hdeem2.2.5-intelmpi-intel2017
```

Load the `scorep-hdeem sync` plugin that is compatible with the Score-P built for the READEX toolsuite, 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
```

If the overhead for hdeem measurements for the application regions is more than a few percent, you need to switch to manual instrumentation of important coarse-granular regions as explained in Section A.5.5.

A.5.5 Manual Instrumentation

If none of the other filtering methods is successful in reducing the overhead to an acceptable level, then manually annotate regions where most of the computation time is spent. You can find these regions with a standard profiler. It is also recommended to instrument the parents of all the significant regions up until the main caller in the hierarchy. This is an optional step which will allow the annotated regions to be used as identifiers for runtime situations.

1. Build the application with additional options to disable compiler instrumentation (`--nocompiler`) and to enable user region instrumentation (`--user`).

2. Manually annotate coarse granular application regions or any other regions that are of interest for tuning using SCOREP_USER_REGION_DEFINE inside the function definition as shown below:

```
SCOREP_USER_REGION_DEFINE( REGION_HANDLE )
SCOREP_USER_REGION_BEGIN( REGION_HANDLE, "REGION_NAME", SCOREP_USER_REGION_TYPE_COMMON )
// application region
SCOREP_USER_REGION_END( REGION_HANDLE )
```

Note: You also have to instrument the `main` routine.

Section A.7.3 presents an example.

A.6   Application Tuning Parameter (ATP) Library

As explained earlier, it is also possible to optionally exploit application level tuning using the READEX tool suite. This requires some additional manual code annotation and instrumentation to pinpoint the parts of the code that can be exploited as application tuning parameters and annotate them with certain API functions.

A.6.1   Instrumentation for ATP library

1. Include the `atplib.h` header file in the source code.

2. Declare the parameter in the source code using `ATP_PARAM_DECLARE` function. Each parameter must contain a unique name, type, default value, and domain name (uses default domain if domain name is `NULL`):

   ```
   ATP_PARAM_DECLARE("PARAM_NAME", ATP_PARAM_TYPE_RANGE, DEFAULT_VALUE, "DOMAIN_NAME");
   ```

   Available ATP parameter types are:

   - `ATP_PARAM_TYPE_RANGE` - defines a range with min, max and step values
   - `ATP_PARAM_TYPE_ENUM` - defines an array of all possible values

3. Add values to the parameter using `ATP_ADD_VALUES`. The second parameter is an array of values added to the parameter, the third parameter is the number of values added.

   ```
   ATP_ADD_VALUES("PARAM_NAME", {1,5,1}, 3, "DOMAIN_NAME");
   ```

   - If parameter type is range, the number of values should be 3 and the values array should contain {`min_value`, `max_value`, `step`}.
   - If the parameter type is enum, then the values array should contain all the possible values that the parameter can have, and the number of values parameter indicates how many values are in this array.

4. Add the call for parameter value assignment. Assigns the parameter value to `control_variable`. The value is assigned by RRL. In case no value is available to RRL, the default parameter value defined in ATP is used:

   ```
   ATP_PARAM_GET("PARAM_NAME", &control_variable, "DOMAIN_NAME");
   ```

5. Add constraint to the parameters of domain `"DOMAIN_NAME"` (**optional**):

   ```
   ATP_CONSTRAINT_DECLARE("CONSTRAINT_NAME", "expr", "DOMAIN_NAME");
   ```

   - The constraint is expressed in the form of a character string `"expr"` which contains a logical expression of how parameters in this domain are constrained (see example in Section A.7.4).

- Any ATP parameters declared in the application can be used in the constraint as long as they belong to the same domain as the constraint.

- Multiple constraints can be defined for the same domain.

- If the domain name is not specified (`NULL`) the constraint will apply to parameters in the default domain.

Section A.7.4 presents an example.

A.6.2 Using the ATP Library

1. Build the application by linking with the ATP library (`-latp`) .

2. Specify a search algorithm for the ATP library from among `exhaustive_atp` and `individual_atp` strategies. This is done by adding sections in the READEX configuration file (`readex_config.xml`) used as input for PTF during DTA as shown below:

```
<periscope>
  <atp>
    <searchAlgorithm>
      <name>exhaustive_atp</name>
      <name>individual_atp</name>
    </searchAlgorithm>
  </atp>
</periscope>
```

For the individial strategy, the *keep* factor is always 1. Updating/extending the READEX configuration file was explained in detail in Section A.3.2.

3. Running the application: there are two phases for running the application with ATP:

- parameter collection phase - parameters, constraints and explorations defined in application are collected and saved for the tuning system to explore.

- parameter exploration phase - declaration functions are turned off and the tuning system can explore the parameter combinations by providing parameter values through the `ATP_PARAM_GET` function.

There are two ATP modes available that allow to enable which phases will be used in the application, although the parameter collection phase needs to be run at least once for the application to allow parameter collection and ATP configuration file creation.

- **DTA mode:**
  - Includes both ATP phases.
  - `ATP_EXECUTION_MODE` environment variable should be set to DTA.
  - It is necessary to run the application in DTA mode at least once in order to generate the `ATP_description_file.json`.

– Starts with parameter collection phase: parameter, constraint and exploration declaration functions are executed only once.

– Second time the same parameter declaration is executed it triggers the end of parameter collection phase, generates `ATP_description_file.json` and begins the exploration phase.

– `ATP_PARAM_GET` assigns parameter values decided by RRL (In the first phase default value is used).

- **RAT mode:**

  – Only parameter exploration phase is running.

  – `ATP_EXECUTION_MODE` environment variable should be unset or set to RAT.

  – Declaration functions are shut down, only `ATP_PARAM_GET` function is working.

  – Details of parameters are loaded from `ATP_description_file.json`.

A.7   Examples

A.7.1   Runtime Filtering

Apply `scorep-autofilter` as follows:

```
scorep-autofilter -t 0.1 -f scorep scorep-*/profile.cubex
```

The file `scorep.filt` contains the region names to be filtered enclosed between `SCOREP_REGION_NAMES_BEGIN` and `SCOREP_REGION_NAMES_END`, as shown below:

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE
Atom::Atom()
Atom::~Atom()
...
SCOREP_REGION_NAMES_END
```

A script to repeat the identification of too fine-granular regions for the miniMD application is available in

```
/projects/p_readextest/miniMD/run_saf.sh
```

and is executed as

```
sh run_saf.sh
```

For different applications, `run_saf.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.

A.7.2   MiniMD Phase Region Annotation

```
void Integrate::run(Atom &atom, Force* force, Neighbor &neighbor,
                    Comm &comm, Thermo &thermo, Timer &timer)
{
  int i, n;
  comm.timer = &timer;
  timer.array[TIME_TEST] = 0.0;
  int check_safeexchange = comm.check_safeexchange;

  mass = atom.mass;
  dtforce = dtforce / mass;
  #pragma omp parallel private(i,n)
  {

    SCOREP_USER_REGION_DEFINE(R1)
```

```
for(n = 0; n < ntimes; n++)
{
  SCOREP_USER_OA_PHASE_BEGIN(R1, "INTEGRATE_RUN_LOOP", 2)

  #pragma omp barrier
  x = &atom.x[0][0];
  v = &atom.v[0][0];
  f = &atom.f[0][0];
  xold = &atom.xold[0][0];
  nlocal = atom.nlocal;

  initialIntegrate();

  #pragma omp barrier
  #pragma omp master
  timer.stamp();

  if((n + 1) % neighbor.every)
  {
    #pragma omp barrier
    comm.communicate(atom);
    #pragma omp master
    timer.stamp(TIME_COMM);
    #pragma omp barrier
  }
  else
  {
    {
      if(check_safeexchange)
      {
        #pragma omp master
        {
          double d_max = 0;
          for(i = 0; i < atom.nlocal; i++)
          {
            double dx = (x[3 * i + 0] - xold[3 * i + 0]);
            if(dx > atom.box.xprd) dx -= atom.box.xprd;
            if(dx < -atom.box.xprd) dx += atom.box.xprd;
            double dy = (x[3 * i + 1] - xold[3 * i + 1]);
            if(dy > atom.box.yprd) dy -= atom.box.yprd;
            if(dy < -atom.box.yprd) dy += atom.box.yprd;
            double dz = (x[3 * i + 2] - xold[3 * i + 2]);
            if(dz > atom.box.zprd) dz -= atom.box.zprd;
            if(dz < -atom.box.zprd) dz += atom.box.zprd;
            double d = dx * dx + dy * dy + dz * dz;
            if(d > d_max) d_max = d;
          }
          d_max = sqrt(d_max);
          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))
            printf("Warning: Atoms move further than your subdomain size, \
                    which will eventually cause lost atoms.\n" \
            "Increase reneighboring frequency or choose a different processor grid\n" \
            "Maximum move distance: %lf; Subdomain dimensions: %lf %lf %lf\n", \
            d_max, atom.box.xhi - atom.box.xlo, \
            atom.box.yhi - atom.box.ylo, \
            atom.box.zhi - atom.box.zlo);
        }
      }

      #pragma omp master
```

```
        timer.stamp_extra_start();
        comm.exchange(atom);
        comm.borders(atom);
        #pragma omp master
        {
          timer.stamp_extra_stop(TIME_TEST);
          timer.stamp(TIME_COMM);
        }
        if(check_safeexchange)
          for(int i = 0; i < 3 * atom.nlocal; i++) atom.xold[i] = atom.x[i];
      }
    #pragma omp barrier
    neighbor.build(atom);

      #pragma omp barrier
      #pragma omp master
      timer.stamp(TIME_NEIGH);
    }
    force->evflag = (n + 1) % thermo.nstat == 0;
    force->compute(atom, neighbor, comm, comm.me);

    #pragma omp master
    timer.stamp(TIME_FORCE);

    if(neighbor.halfneigh && neighbor.ghost_newton)
    {
      comm.reverse_communicate(atom);

      #pragma omp master
      timer.stamp(TIME_COMM);
    }
    v = &atom.v[0][0];
    f = &atom.f[0][0];
    nlocal = atom.nlocal;

    #pragma omp barrier
    finalIntegrate();

    #pragma omp barrier
    if(thermo.nstat) thermo.compute(n + 1, atom, neighbor, force, timer, comm);

  SCOREP_USER_OA_PHASE_END(R1)
    }
  } //end OpenMP parallel
}
```

This example is also available on Taurus in

```
/projects/p_readextest/miniMD/integrate.cpp
```

A.7.3   Manual Instrumentation

```
main()
{
  ...
  integrate.run(...);
  ...
```

```
}

void Integrate::run(...)
{
  SCOREP_USER_REGION_DEFINE( REGION_HANDLE )
  SCOREP_USER_REGION_BEGIN( REGION_HANDLE, "REGION_NAME", SCOREP_USER_REGION_TYPE_COMMON )
  // application region
  SCOREP_USER_REGION_END( REGION_HANDLE )
}
```

Example For the miniMD application, manually annotate
`ForceLJ::compute_halfneigh()` and its parents `Integrate::run()` and `main()` as
significant regions as shown in the following files respectively:

```
/projects/p_readextest/miniMD/force_lj.cpp
/projects/p_readextest/miniMD/integrate.cpp
/projects/p_readextest/miniMD/ljs.cpp
```

A.7.4   Application Tuning Parameter (ATP) Instrumentation

```
void foo(){
    int atp_cv;
    ...
    ATP_PARAM_DECLARE("solver", ATP_PARAM_TYPE_RANGE, 1, "DOM1");
    int solver_values[3] = {1,5,1};
        //{1,5,1} means a range with a minimum value of 1, a maximum one of 5 and an increment of 1
    ATP_ADD_VALUES("solver", solver_values, 3, "DOM1");
    ATP_PARAM_GET("solver", &atp_cv, "DOM1");

    switch (atp_cv){
      case 1:
          // choose algorithm 1
          break;
      case 2:
          // choose algorithm 2
          break;
      ...
  }

    int atp_ms;
    ATP_PARAM_DECLARE("mesh", ATP_PARAM_TYPE_RANGE, 40, "DOM1");
    int mesh_values[3] = {0,120,10};
    ATP_ADD_VALUES("mesh", mesh_values, 3, "DOM1");
    ATP_PARAM_GET("mesh", &atp_ms, "DOM1");
    ATP_CONSTRAINT_DECLARE("const1", "(solver = 1 && 0 <= mesh 40) ||
                                      (solver = 2 && 50 <= mesh <= 80) ||
                                      (solver > 2 && mesh = 120)", "DOM1")
    if ( (atp_ms > 1) && (atp_ms <= 40) ) {
      // algorithm for mesh size 1
      }
    if ( (atp_ms > 40) && (atp_ms <= 80) ) {
      // algorithm for mesh size 2
      }
    if ( atp_ms == 120 ) {
      // algorithm for mesh size 3
```

```
    }
```

A.7.5   Tuning Potential Analysis

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 10 -v 10 -w 10 scorep-<xyz>/profile.cubex
```

   Here, `readex-dyn-detect` takes the granularity for the region as 1 ms with `-t 0.001`. The option `-p INTEGRATE_RUN_LOOP` is given to the tool to identify the phase region from the `profile.cubex` call tree. The three options `-c 10 -v 10 -w 10` define thresholds for the compute intensity variation (absolute value), time deviation in % of the mean region time and weight of the region (%) which is execution time w.r.t. phase time.

A script to perform steps 1 and 2 for the miniMD application is available in

```
/projects/p_readextest/miniMD/run_rdd.sh
```

and is executed as

```
sh run_rdd.sh
```

For different applications, `run_rdd.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`.
The following lines are printed as part of the output by `readex-dyn-detect` for miniMD:

```
1  ...
2  Significant regions are:
3
4  void Comm::borders(Atom&)
5  void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [with int EVFLAG = 0; int GHOST_NEWTON = 1]
6  void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [with int EVFLAG = 1; int GHOST_NEWTON = 1]
7  void Neighbor::build(Atom&)
8
```

```
 9  |
10  | Significant region information
11  | ==============================
12  | Region name              Min(t)        Max(t)      Time Dev.(%Reg) Ops/L3miss    Weight(%Phase)
13  |
14  | void Comm::borders(Atom&)   0.001         0.001         2.6            109             0
15  | void ForceLJ::compute_hal   0.013         0.014         2.9             97            68
16  | void ForceLJ::compute_hal   0.016         0.016         0.0             91             1
17  | void Neighbor::build(Atom   0.047         0.048         0.7            332            23
18  |
19  |
20  | Phase information
21  | =================
22  | Min               Max               Mean            Dev.(% Phase)      Dyn.(% Phase)
23  |
24  | 0.0138626         0.0664566         0.020337         72.731             258.612
25  |
26  | ...
27  |
28  | SUMMARY:
29  | ========
30  |
31  | Inter-phase dynamism due to variation of the execution time of phases
32  |
33  | No intra-phase dynamism due to time variation
34  |
35  | Intra-phase dynamism due to variation in the compute intensity of the following important significant
36  |   regions
37  |
38  | void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [with int EVFLAG = 0; int GHOST_NEWTON = 1]
39  |
40  | void Neighbor::build(Atom&)
```

The printed output above for the miniMD application can be divided into three parts:

First, lines 2–7 list the names of the significant regions computed from the detection algorithm. For details of the algorithm, please see deliverable D2.1 [7].

Secondly, lines 10–26 show the profile statistic output for the detected significant regions and phase region. This section consists of two parts. The significant region information presents the minimum and the maximum of the execution time for each significant region as well as the aggregated execution time for the region. It also prints the time deviation in % with respect to its mean value. The Ops/L3miss column prints the absolute compute intensity value. In the last column, Weight(%Phase), is the execution time with respect to phase time.

After that, the tool summarises the statistics information for the phase region. It shows the minimum, maximum, and mean values of the execution time spent on the phase region as well as the aggregated execution time for the phase. The Dev.(% Phase) column prints the time deviation w.r.t. the phase mean execution time. The last column, Dyn.(% Phase), prints the variation between minimum and maximum execution time w.r.t. the mean execution time of the phase.

Finally, the tool prints the summary results of the dynamism analysis (lines 28–40). First, if the standard deviation of the phase is larger than the variation threshold, then the tool indicates having inter-phase dynamism due to variation of the execution time of phases.

Otherwise, the application does not have inter-phase dynamism. For miniMD, the variation is larger than the threshold. So the tool detects inter-phase dynamism for miniMD.

The tool compares `Weight(%Phase)` with the given threshold given by the user. If a significant region has enough weight and its time deviation w.r.t. region is more than the time deviation threshold given via `-v`, the tool detects intra-phase dynamism for these significant region(s) due to time variation. For miniMD, there are two significant regions having weights larger than the given threshold ($> 10\%$):

```
void ForceLJ::compute_halfneigh(Atom&, Neighbor&, int) [ with int EVFLAG = 0; int GHOST_NEWTON = 1 ]
void Neighbor::build(Atom&)
```

But neither of them has a time deviation greater than 10%. So the tool does not detect intra-phase for miniMD due to time deviation.

The tool computes the variation of the compute intensity for the set of detected significant regions having a minimum weight of 10%. For miniMD the variation value is larger than the provided threshold of compute intensity specified with `-c`. So the tool detects intra-phase dynamism due to the variation in the compute intensity characteristic and lists the region names that exhibit intra-phase dynamism.