READEX Tool Suite – User Guide

Contents

1 Introduction

The READEX tool suite has been developed to optimise the execution of HPC applications on Exascale systems for energy efficiency. This is achieved by analysing and modelling the dynamic behaviour of an HPC application in a step known as *design-time analysis*. This step results in identifying the optimal values for different hardware, system software and application parameters that are available for tuning. Currently, READEX is capable of tuning two hardware parameters (CPU core and uncore frequencies), a system software parameter (number of OpenMP threads) and application parameters that have been exposed by the user of the tool suite or the application owner. Following this, the optimal values that have been identified for different regions of an application are set during the production run of the application in a step known as *runtime application tuning*.

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

1. Instrument the application with Score-P. (Section 2)
2. Perform design-time analysis of application to create tuning model. (Section 3)
3. Use the tuning model during the production run of the application for runtime tuning. (Section 4)

For a better understanding of the READEX approach we recommend to have a look at Deliverable D2.3. The tools in the READEX tool suite are accessible by installing the tool suite as explained in the READEX website.

2 Application instrumentation

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, by 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 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.

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. 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.1 and A.2, respectively.

2. You may also remove MPI and OpenMP region instrumentation overhead as described in Section A.3.

3. Then switch on the energy measurements which (sometimes) may cause significant performance overhead. Verify the overhead again.
   For instance, RAPL can be used for energy measurement, which has less overhead than HDEEM. Note that the energy measurements from RAPL may not be precise enough. For instance, an execution 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.

Do not proceed to energy tuning if the overhead is too high, e.g. more than 5%.

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.

Example The `for-loop` body in `Integrate::run()` is annotated as a phase region as shown in the example in Section D.3.

2.4 Application tuning parameter instrumentation

As mentioned earlier, the READEX tool suite also allows tuning application parameters. This is optional and the application parameters to be tuned should be identified and exposed by the tool suite user and/or by the application owner. 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 B.1.

3 Design-time Analysis (DTA)

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 2.3.

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

1. Build the application with instrumentation by `scorep`. Add `--online-access --user` for the manually annotated phase region.

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

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

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

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

```
readex-dyn-detect -p <phase_region_name>
                  [-t <region granularity threshold in sec>]
                  [-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>]
                  <path_to_cubex_file>/profile.cubex
```

   If the readex configuration file already exists, only the list of significant regions will be updated according to the outcome of `readex-dyn-detect`. It the file is missing, the template file will be copied from the installation directory and updated with the significant regions.

   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). (default 0.1)

   -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. (default 10%)

   -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. (default 10%)

   -w This threshold specifies the minimal weight of a region such that any dynamism due to time variation or compute intensity variation is reported. (default 10%)

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

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.

READEX distinguishes between two types of dynamism in an application:

- Inter-phase dynamism: This occurs when each phase (execution instance) of a phase region in the application exhibits different characteristics. This results in different values for the measured objective values and thus may require different configurations to be applied for the tuning parameters.
- Intra-phase dynamism: This occurs when each runtime situation (execution instance) of the significant regions in a phase region exhibits different characteristics and thus may need different configurations to be applied for the tuning parameters.

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 D.6 presents an example.

3.2 Configure DTA

The next step of the DTA is to update the READEX configuration file (`readex_config.xml`) that the `readex-dyn-detect` tool has generated 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: READEX currently supports tuning two hardware parameters (processor core frequency and uncore frequency) and a system software parameter (number of OpenMP threads). A minimum of one of these tuning parameters must be specified. Specify the ranges (minimum, maximum, step size, and default) for the processor core frequency and the uncore frequency in MHz. The default frequencies are given by the system and are defined during the installation. Appendix C describes how to retrieve the available CPU core and uncore frequencies in a system.

   For OpenMP threads, specify the lower bound and the step size to increment to the next value. The upper bound is given by the `psc_frontend` command.

Example

```
<tuningParameter>
  <frequency>
     <min_freq>1200</min_freq>
     <max_freq>2400</max_freq>
     <freq_step>200</freq_step>
```

```
    <default> 2500</default>
  </frequency>
  <uncore>
    <min_freq>1000</min_freq>
    <max_freq>3000</max_freq>
    <freq_step>200</freq_step>
    <default> 3000</default>
  </uncore>
  <openMPThreads>
    <lower_value>1</lower_value>
    <step>2</step>
  </openMPThreads>
</tuningParameter>
```

Note that application parameters may be optionally tuned as introduced in Section 2.4 and explained in Section B.1.

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. It expresses the energy consumption per instruction and can be used for applications where the amount of computation in a phase varies without changing the phase characteristics ("more of the same"). 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.

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

3. Specify the energy metrics: Specify the energy plugin name and associated metric names. For `hdeem_sync_plugin`, it is possible to measure the energy for the whole node or/and two CPUs respectively. 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>
  </metrics>
</periscope>
```

To specify the EXAMON energy plugin `examon_sync_plugin`, use the configuration as follows:

Example

```
<periscope>
  <metricPlugin>
    <name>examon_sync_plugin</name>
  </metricPlugin>
  <metrics>
    <node_energy>EXAMON/BLADE/E</node_energy>
  </metrics>
</periscope>
```

4. Specify a search algorithm: Specify a search algorithm from exhaustive, random or individual. For the random search strategy, specify the number of samples (scenarios) that the plugin should limit to. For the individual search, specify the number of tuning parameter values to *keep* in the search space. The search algorithm should be specified under `<periscope> </periscope>`.

Example

```
<periscope>
  <searchAlgorithm>
    <name>exhaustive</name>
    <name>random</name>
      <samples>2</samples>
    <name>individual</name>
      <keep>2</keep>
  </searchAlgorithm>
</periscope>
```

The search algorithm together with the tuning parameter specification determines the tuning time. Exhaustive search leads to the biggest number of configurations that are tested in subsequent program phases. The individual strategy reduces the number significantly since not the cross product of all tuning parameters is investigated, but the parameters are optimized independently. With the random strategy, the number of experiments can be specified.

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 B.2.

3.3 Tuning application configuration parameters

The READEX tool suite provides an additional optional step before generating the tuning model. Applications frequently have *Application Configuration Parameters (ACP)* that might affect the performance and energy consumptions. These parameters can be tuned for the given objective with PTF. It provides a tuning plugin which takes a specification of the ACPs with their possible values and searches for the best settings by restarting the application for each experiment.

The tuning plugin is `readex_configuration`. It relies on a configuration file specifying the input files, the configuration parameters and their possible values. The plugin first reads the configuration files and then generates the possible configurations based on the defined search algorithm. For each configuration, the template file of an input file is copied, the configuration variables are replaced by the selected values, and the application is restarted.

Note that names of configurations variables may be composed of the letters ’a’-’z’, ’A’-’Z’, ’0’-’9’, ’_’, ’-’, ’ ’, ’*’, ’@’, ’,’, ’(’, ’)’, ’ ’, ’.’, ’/’ and ’=’.

Be aware that only a single phase is executed for each experiment which significantly reduces the tuning time. In case you want to omit some phases before starting the measurements or you want it to be based on multiple subsequent phases, please use the optional PTF arguments `--delay` and `--iterations`. At the end of the search, the optimal configuration is written into the input files and output into the configured results file.

The tuning plugin and its configuration file are selected via command line arguments as shown in the example below. Both configuration files are to be given, i.e., `readex_config.xml` and `appConfigParams.cfg`.

```
...

psc_frontend --apprun=....
                  --config-file=readex_config.xml
                  --tune=readex_configuration --readex-app-config=appConfigParams.cfg
...
```

The plugin takes from `readex_config.xml` the configuration for the energy measurements, all other configurations are taken from the plugin specific configuration file. The syntax of the specification in the plugin specific configuration file is given below:

```
Specification: CONFIGURATION ConfigList APPLICATIONPARAMETERS FileList
             | APPLICATIONPARAMETERS FileList
//-----------------------------------------
//Specification of application configuration parameters (ACP)
```

```
//First specify the input file with its template file
//PTF copies the template file to the input file and
//replaces the ACP name with the current value
//-------------------------------------------

FileList:          FileSpec FileList | FileSpec
FileSpec:          FileSpecification  TemplateFileSpecification ParameterList
FileSpecification: FILENAME STRING ’;’
TemplateFileSpecification: TEMPLATEFILENAME STRING ’;’

//-------------------------------------------
//In each input file a number of ACP can be provided
//For each ACP specify the name and the values.
//The first value is taken as default.
//-------------------------------------------

ParameterList:     ParameterSpecification ParameterList | ParameterSpecification
ParameterSpecification: PARAMETER STRING VALUESSTR TpRange ’;’
TpRange:           IntRange | StringRange

IntRange:          ’[’ INT ’,’ INT ’]’ | ’[’ INT ’,’ INT ’,’ INT ’]’
StringRange:       ’[’ StringList ’]’
StringList:        STRING ’,’ StringList | STRING
//STRING:            [’-’ ’a’-’z’ ’A’-’Z’ ’0’-’9’ ’_’
//                    ’~’ ’*’ ’@’ ’,’ ’(’ ’)’ ’ ’ ’.’ ’/’ ’=’ ]+


//-------------------------------------------
//Specification of general configuration parameters
//-------------------------------------------

ConfigList:        ConfigStmt ConfigList | ConfigStmt

ConfigStmt:        SearchAlgSpecification ’;’
                   | IndividualKeepSpecification ’;’
                   | SampleCountSpecification ’;’
                   | ResultsFileSpecification ’;’
                   | ObjectiveSpecification ’;’

SearchAlgSpecification:      SEARCHALG STRING
IndividualKeepSpecification: INDIVIDUALKEEP INT
SampleCountSpecification:    SAMPLECOUNT ’=’ INT

ResultsFileSpecification:    RESULTSFILE STRING

ObjectiveSpecification:      OBJECTIVE ObjectiveName
ObjectiveName:     ENERGY| NORMALIZEDENERGY
                   | CPUENERGY | NORMALIZEDCPUENERGY
                   | EDP | NORMALIZEDEDP
                   | ED2P | NORMALIZEDED2P
                   | TIME | NORMALIZEDTIME
                   | TCO | NORMALIZEDTCO

```

The syntax specification follows the Backus-Naur-Form as it is used in `bison`. Terminal symbols are capitalized. The special sign `’|’` indicates an alternative. Thus, for example, a list of configurations is simply a concatenation of individual configuration statements. All statements end with a semicolon.

Here is an example:

```
    configuration
        results_file "optimal_configuration.txt";
        search_algorithm "individual";
        individual_keep 1;
        objective Energy;
    application_parameters
        input_file "info.txt";
        template_file "info.template";
        parameter "P1" values [ "1", "2"];
        parameter "P2" values [ "z7-@3", "x8(23).txt"];
        input_file "info1.txt";
        template_file "info1.template";
        parameter "P11" values [ "3", "4"];
```

Outcome: The results file containing the optimal configuration for the ACPs and the input files where the ACPs were replaced by the optimal setting.

3.4 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 2.3 (`scorep --user --online-access`) 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 requested to at least 2, and allocate enough memory to fit the application. 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 substrate plugin and parameter control plugins compatible with Score-P and PTF.

4. Load the Score-P plugin to be used for energy measurements and set the corresponding Score-P environment variables. The `SCOREP_SUBSTRATE_PLUGINS` variable is used to specify the substrate plugins to be used, which for READEX should include `rrl`. The `SCOREP_RRL_PLUGINS` are a comma-separated list of parameter tuning plugin names to be used by RRL (`cpu_freq_plugin`, `uncore_freq_plugin`, `OpenMPTP`, etc.). The `SCOREP_METRIC_PLUGINS` is used to set the metric plugin used to perform energy measurements.

5. The `SCOREP_RRL_CHECK_IF_RESET` variable sets the behaviour of the settings stack of the configuration manager. Possible values are `reset` (default, every change will be saved on the settings stack), and `no_reset` (only the default and current values of parameters will be saved; new parameter values overwrites the current values).

6. Apply PTF on the application with the `psc_frontend` command. Specify the instrumented phase region name for the option `--phase` and the readex configuration file for `--config-file`. Specify the `readex_intraphase` plugin for `--tune`.

   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.

If `readex-dyn-detect` reports inter-phase dynamism for the application, the user is advised to proceed with inter-phase tuning, as described in Section 3.5. Note that the ATPs are disabled for inter-phase tuning.

```
### Batch system (SLURM, PBS, ...) directives ###
### number of nodes requested (at least 2); 1 for PTF and remaining for application run
### ...
### memory requested for application run
### ...
###

###
### Load READEX tool suite modules, or set paths to executables and libraries
###

echo "run PTF begin."

export SCOREP_SUBSTRATE_PLUGINS=rrl
export SCOREP_RRL_PLUGINS=<list of RRL tuning plugins>
export SCOREP_RRL_VERBOSE="WARN"
export SCOREP_RRL_CHECK_IF_RESET=<reset|no_reset>

# Energy plugin name: for eg. hdeem_sync_plugin, x86_energy_sync_plugin, examon_sync_plugin>
export SCOREP_METRIC_PLUGINS=<Energy Plugin Name>

export SCOREP_METRIC_PLUGINS_SEP=";"

###
### set Score-P metric energy plugin environment variables; for eg. SCOREP_METRIC_HDEEM_SYNC_PLUGIN_*
###

export SCOREP_MPI_ENABLE_GROUPS=ENV

psc_frontend --apprun="<application executable and command-line arguments>"
             --mpinumprocs=<number of MPI processes>
             --ompnumthreads=<maximum number of OpenMP threads>
             --phase=<phase region name>
             --tune=<readex_intraphase | readex_interphase>
             --config-file=<READEX configuration file name; for eg. readex_config.xml>
             --info=<2 .. 7>
             --selective-info=AutotuneAll,AutotunePlugins

echo "run PTF done."
```

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.

3.5 Inter-Phase Tuning

To exploit the inter-phase dynamism, only the following must be changed in addition to the steps described in Sections 3.2 and 3.4.

Configure DTA For inter-phase tuning, the search algorithm specified by the user is ignored, and the plugin sets the `random` search strategy by default. Specify the number of experiments or points to analyze during clustering under the `samples` tag. Higher values enable a better analysis of the dynamism between the phases and result in a more refined tuning model.

Tuning Model Creation Specify the readex_interphase plugin for `--tune`.

Outcome The readex_interphase tuning plugin clusters similar phases using the DBSCAN algorithm, and determines the best configuration for each cluster, which will be applied to all the phases belonging to that cluster. It also determines the best configuration for individual rts's in the cluster.

- A printed summary of the created scenarios, the properties found in each scenario, the measured objective values for the phase in each scenario, per-cluster results showing the optimum scenario for all the phases of the cluster as well as the best configuration for each rts of the cluster, 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 clusters generated by the clustering algorithm, the set of phases belonging to each cluster, the ranges of the features that were used for clustering, 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.

3.6 Visualization of the Tuning Model

The visualization tool of the tuning model is constructed based on the JavaScript library D3.js. The tool can be built as an app for macOS, Linux, and Windows, using Electron Packager. The latest source of the visualization tool of the tuning model can be cloned by `git clone https://periscope.in.tum.de/git/visualizationOfTM.git`. This source already contains the source for *Electron Packager* and is configured to be build as a standalone application.

Electron Packager is a command-line tool and a Node.js library. It bundles Electron-based application source code (with a renamed Electron executable and supporting files) into folders ready for distribution.

Mandatory Library: The tool requires the `Node.js` library to be installed. The library can be downloaded from `https://nodejs.org`.

Usage: The tool requires two mandatory files which are generated at the end of DTA. The files are: `tuning_model.json` and `rts.xml`. The first one is the tuning model file and the latter contains the execution time information of the rts's. The tool can be started with the following command in the source directory:

```
npm start
```

First, select the tuning model file `tuning_model.json` and the file with the rts's `rts.xml`. You have to select both files in the dialog. The tool checks the extension and assumes that a file with `.json` is the tuning model and the extension `.xml` identifies the rts file.

The tool will then generate the forced layout view of the tuning model. This will look as in Figure 1. For each scenario a circle is generated. The size of the circle represents the weight, i.e., the aggregated execution times of all rts's of the scenario as percent of the phase time. The

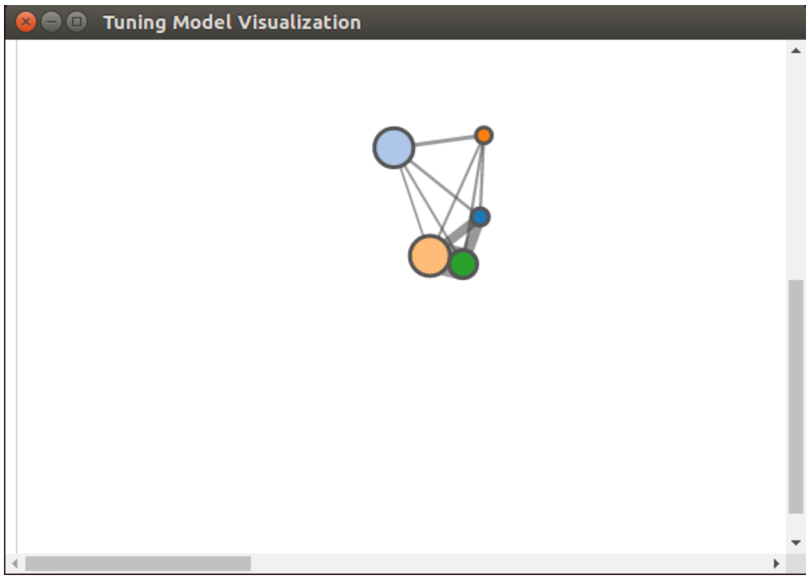


Figure 1: Forced layout graph of the tuning model

thickness of the lines represents the similarity of two connected scenarios. Hovering over a circle triggers a tool tip that gives detailed information about the scenario. Clicking on a scenario opens individual circles for each rts in this scenario.

More details about the tool can be found in deliverable D2.2.

4 Runtime Application Tuning (RAT)

4.1 Production Run with Tuning Model

The following steps describe how to use RRL to tune the application during its production run.

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

2. For the application run tuned with RRL, use the application built for analysis with PTF as described in Section 3. If the readex_interphase tuning plugin was used for DTA, perform the following steps before proceeding with RAT:

   - To perform runtime tuning after the inter-phase analysis, the cluster prediction library must be invoked. The `SCOREP_USER_PARAMETER` macro must be added to the source code, with a call to the `predict_cluster` function of the cluster prediction library.

     For C/C++ applications:

     ```
     #include <cluster_prediction.h>
     // loop starts
     SCOREP_USER_OA_PHASE_BEGIN( REGION_HANDLE, "PHASE_REGION_NAME", SCOREP_USER_REGION_TYPE_COMMON )
     SCOREP_USER_PARAMETER_INT64( "Cluster", predict_cluster() )
     // loop body (phase region)
     SCOREP_USER_OA_PHASE_END( REGION_HANDLE )
     // loop ends
     ```

     For Fortran applications:

     ```
     SCOREP_USER_REGION_DEFINE( REGION_HANDLE )
     SCOREP_USER_PARAMETER_DEFINE(cluster)
     // loop starts
     SCOREP_USER_OA_PHASE_BEGIN( REGION_HANDLE, "PHASE_REGION_NAME", SCOREP_USER_REGION_TYPE_COMMON )
     SCOREP_USER_PARAMETER_INT64( "Cluster", predict_cluster() )
     // loop body (phase region)
     SCOREP_USER_OA_PHASE_END( REGION_HANDLE )
     // loop ends
     ```

   - Add the linker flags in the Makefile and rebuild the application.

3. Set the number of nodes to run the application, and allocate enough memory to fit the application. Here, the number of nodes required is the same as the number of nodes on which to run the application.

4. For the RRL-tuned run of the application perform the following steps:

   (a) Disable Score-P profiling and tracing, set the Score-P substrate plugins to `rrl`, RRL plugins to the tuning plugins to use and the tuning model to the file generated by PTF.

   (b) The `SCOREP_RRL_CHECK_IF_RESET` variable sets the behaviour of the settings stack of the configuration manager. Possible values are `reset` (default, every change will be saved on the settings stack), and `no_reset` (only the default and current values of parameters will be saved; new parameter values overwrites the current values).

   (c) Depending on whether the calibration mechanism is enabled or disabled for RRL, set the environment variable `SCOREP_FORCE_CFG_FILES` to `true` or `false`, respectively, to enable or disable generation of Score-P configuration file for use by RRL.

   (d) Run the RRL-tuned version of the application using the application executable built for PTF.

```
### Batch system (SLURM, PBS, ...) directives ###
### number of nodes requested for application run
### ...
### memory requested for application run
### ...
###

###
### Load READEX tool suite modules, or set paths to executables and libraries
###

### start RRL-tuned run
export SCOREP_ENABLE_PROFILING="false"
export SCOREP_ENABLE_TRACING="false"
export SCOREP_SUBSTRATE_PLUGINS="rrl"
export SCOREP_RRL_PLUGINS=<list of RRL tuning plugins>
export SCOREP_RRL_TMM_PATH=<tuning model file name>
export SCOREP_RRL_CHECK_IF_RESET=<reset|no_reset>
export SCOREP_MPI_ENABLE_GROUPS=ENV

### If RRL calibration mechanism is enabled, then set SCORE_FORCE_CFG_FILES to true.
### This will create Score-P configuration files in a Score-P directory for the application run.
### If RRL calibration mechanish is disabled, then set SCORE_FORCE_CFG_FILES to false.
export SCOREP_FORCE_CFG_FILES=false

### run RRL-tuned application
# code to start application run with command-line arguments
### end RRL-tuned run
```

4.2 Visualise Configuration Switching

There are two ways for visualising the configuration switching:

1. The first is a Score-P metric plugin which is implemented as part of RRL and it shows the RRL perspective to the switching, i.e. the configurations applied by RRL. It can be used during DTA and RAT.

2. The other way is to use the Score-P APAPI metric plugin to trace metrics such as `APAPI_TOT_CYC` to view the processor core frequency and metric plugins such as `upe_plugin` to view the uncore frequency. These metrics show what actually happens in the processor. They can just be applied during RAT.

By comparing the output of both visualization ways mentioned above, it can be easily verified if the settings applied by RRL correspond to what actually happened in the processor.

Using the Score-P metric plugin for viewing the configuration set by RRL When tracing is enabled, Score-P generates OTF2 traces which can be viewed in Vampir. Hence, to get the metrics in trace, the first step is to enable tracing as follows.

```
export SCOREP_ENABLE_TRACING=true
```

1. Set the environment variable `SCOREP_METRIC_PLUGINS` to specify the metric plugin for visualization of tuning parameters settings in Vampir.

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

2. Set the environment variable `SCOREP_METRIC_SCOREP_SUBSTRATE_RRL` to specify the tuning parameters which need to be added to trace. For the hardware and software tuning parameters, names of the parameter control plugins (PCPs) are used. The list of tuning parameters

Tuning Parameter	**Parameter Control Plugin**
CPU Frequency	cpu_freq_plugin
Uncore Frequency	uncore_freq_plugin
Number of Threads Scheduling Type Schedule chunk size	OpenMPTP
Energy Performance Bias (EPB)	epb_plugin
MPIR_CVAR_REDUCE_SHORT_MSG_SIZE	mpit_plugin

Table 1: Tuning Parameters and their correspoding PCPs

with their corresponding PCPs are listed in Table 1. If the user wants to load all the hardware and software tuning parameters, it can be done by simply setting the environment variable to `*`.

Application Tuning Parameters (ATPs) 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. 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 metric plugin and request the processor core frequency and uncore frequency to be traced can be set as follows:

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

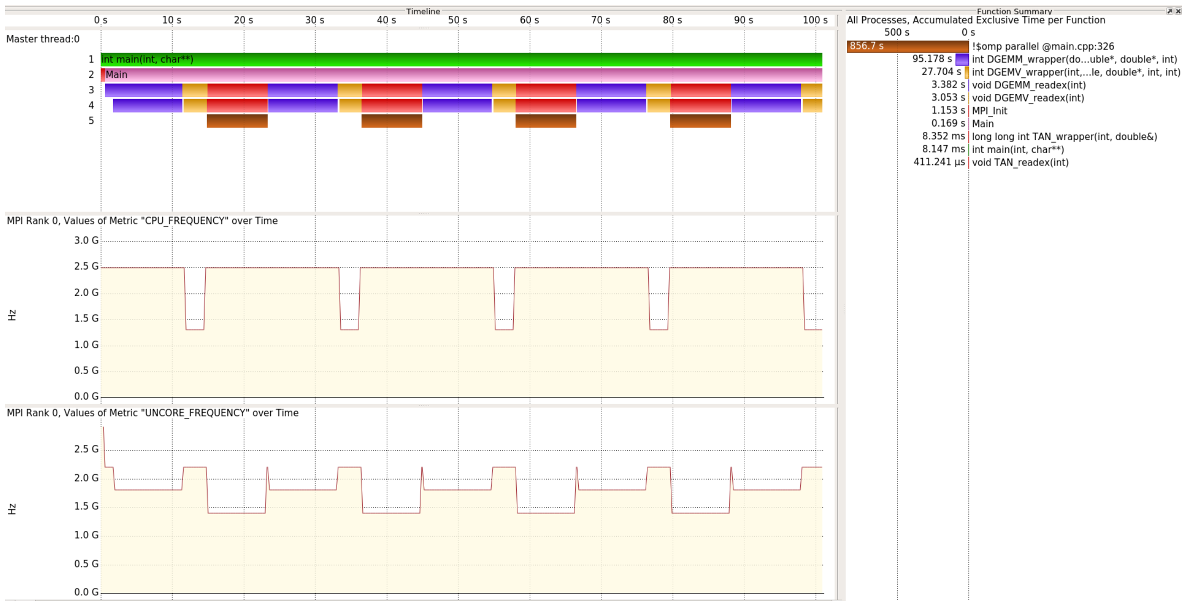
An example trace showing the different configurations of core and uncore frequency applied during RAT is given in Figure 2. The Score-P tracing functionality is enabled and the metric plugin is employed during the RAT phase for Blasbench benchmark. The tuning parameters in Figure 2 are named as CPU_FREQUENCY and UNCORE_FREQUENCY. Figure 2 shows the configurations which have been applied at runtime. To confirm that these configurations are actually set in the processor, Score-P APAPI and uncore metric plugins, which are explained next, can be used.

Using the Score-P APAPI and uncore metric plugins To use the APAPI and uncore metric plugins, and to trace 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 generated trace file will be placed in the folder specified by the environment variable `SCOREP_EXPERIMENT_DIRECTORY`. This can be viewed using Vampir.



All Processes, Accumulated Exclusive Time per Function

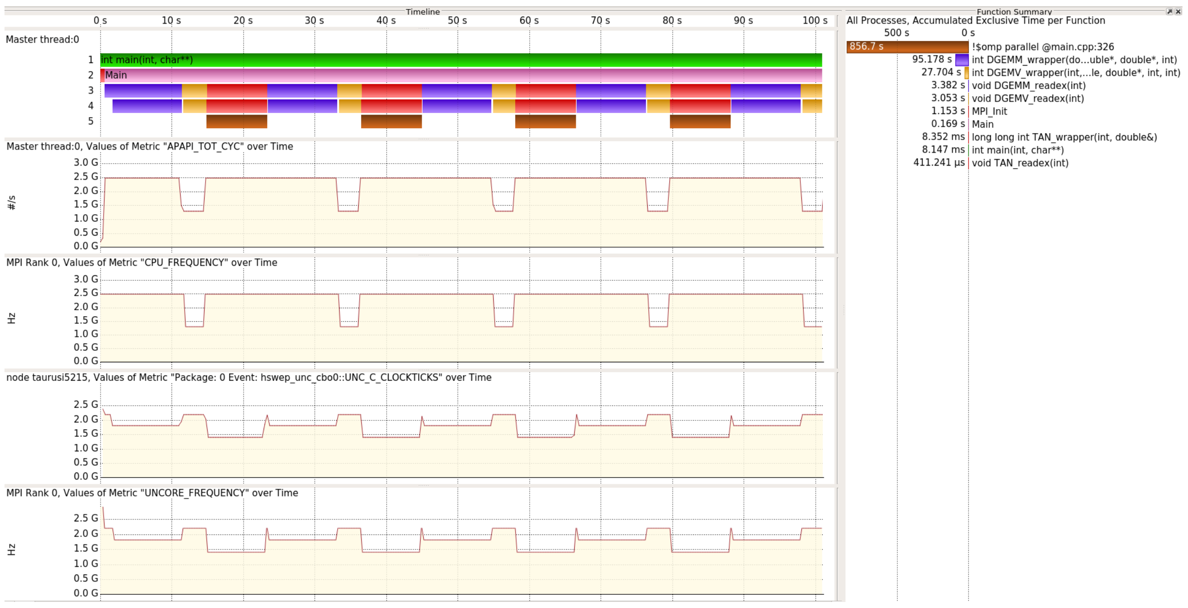
856.7 s	!$omp parallel @main.cpp:326
95.178 s	int DGEMM_wrapper(do...uble*, double*, int)
27.704 s	int DGEMV_wrapper(int,...le, double*, int, int)
3.382 s	void DGEMM_readex(int)
3.053 s	void DGEMV_readex(int)
1.153 s	MPI_Init
0.169 s	Main
8.352 ms	long long int TAN_wrapper(int, double&)
8.147 ms	int main(int, char**)
411.241 µs	void TAN_readex(int)

Figure 2: Vampir trace showing the different configurations of CPU_FREQUENCY and UNCORE_FREQUENCY applied at runtime for the Blasbench benchmark

Figure 3 shows the trace for the APAPI_TOT_CYC and UNC_C_CLOCKTICKS traced using the APAPI and uncore metric plugins respectively along with the tuning parameters CPU_FREQUENCY and UNCORE_FREQUENCY shown in Figure 2.

Both the traces presented in Figure 2 and Figure 3 are obtained in the same run of Blasbench benchmark. It can be easily visualized in Figure 3 that the APAPI_TOT_CYC trace confirms the trace of CPU_FREQUENCY. The UNC_C_CLOCKTICKS trace in Figure 3 also confirms that the value of UNCORE_FREQUENCY is set as instructed by RRL.

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.


Timeline
0 s
10 s
20 s
30 s
40 s
50 s
60 s
70 s
80 s
90 s
100 s
Master thread:0
1
2
3
4
5
int main(int, char**)
Main
Master thread:0, Values of Metric "APAPI_TOT_CYC" over Time
#/s
3.0 G
2.5 G
2.0 G
1.5 G
1.0 G
0.5 G
0.0 G
MPI Rank 0, Values of Metric "CPU_FREQUENCY" over Time
Hz
node taurusi5215, Values of Metric "Package: 0 Event: hswep_unc_cbo0::UNC_C_CLOCKTICKS" over Time
MPI Rank 0, Values of Metric "UNCORE_FREQUENCY" over Time
Function Summary
500 s
0 s

All Processes, Accumulated Exclusive Time per Function

856.7 s	!$omp parallel @main.cpp:326
95.178 s	int DGEMM_wrapper(do...uble*, double*, int)
27.704 s	int DGEMV_wrapper(int,...le, double*, int, int)
3.382 s	void DGEMM_readex(int)
3.053 s	void DGEMV_readex(int)
1.153 s	MPI_Init
0.169 s	Main
8.352 ms	long long int TAN_wrapper(int, double&)
8.147 ms	int main(int, char**)
411.241 μs	void TAN_readex(int)

Figure 3: Vampir trace showing the APAPI_TOT_CYC and UNC_C_CLOCKTICKS recorded using the Score-P APAPI and uncore plugin respectively

A Filtering and Manual Instrumentation

A.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 instrumented. 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 instrumentation overhead that they wish to retain for the analysis of regions in the application. The higher the threshold value, the lower will be the instrumentation overhead, but also the number of regions accessible to the READEX tool suite.

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 execution 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 D.2 presents an example.

A.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 users do this whenever possible since each existing instrumentation interrupts the program flow during its execution.

In order to apply compile time filtering using the Intel compiler, an additional option needs to specified for `scorep-autofilter`:

```
scorep-autofilter -t <region_granularity_threshold_in_sec>
                  [-f <filter_file_name_without_extension>]
                      [-i <intel_filter_file_name_without_extension>]
                  <path_to_cubex_file>/profile.cubex
```

This will create a filter file that can be used by the Intel compiler to suppress instrumentation of regions. This filter file can then be passed to the Intel compiler using the option `-tcollect-filter=<intel_filter_file_name>`.

A.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. In this case, 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 measurements for all MPI routines except `MPI_Init`, `MPI_Finalize` and other environment routines. These are required for Score-P to work correctly.

A.4 Energy Measurements

Due to potential overhead of energy measurements for application profiling with Score-P, it is necessary to check the overhead when the energy measurements are switched on. Energy measurement on clusters can be done using tools such as HDEEM, RAPL or AMD APM.

If the energy measurement overhead for the application regions is observed to be more than a few percent, you need to reduce measurement and instrumentation even further. A recommended way for this is by using manual instrumentation as explained in Section A.5.

A.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 D.4 presents an example.

B 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. Note that the ATP library can only be used to exploit intra-phase dynamism.

B.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. This 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 D.5).
   - 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 D.5 presents an example.

B.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 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 phase 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.
     - The name and location of the ATP description file can be set by the `ATP_DESCRIPTION_FILE` environment variable. If this variable is not set then the ATP description file will be created in the current working directory as `ATP_description_file.json`.
     - Starts with parameter collection phase: parameter, constraint and exploration declaration functions are executed only once.
     - The second time the same parameter declaration is executed, it triggers the end of the parameter collection phase, generates the ATP description file and begins the exploration phase.
     - `ATP_PARAM_GET` assigns parameter values decided by RRL. In the first phase, the default value is used.
   - **RAT mode:**
     - Only the parameter exploration phase is running.

- The `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 the ATP description file.

C Retrieving CPU Frequencies

In Linux-based systems,

- the available CPU core frequencies can be retrieved as follows:

  ```
  cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
  ```

  For Intel processors, make sure that intel_pstate is not loaded.

- the minimum and maximum values for the CPU uncore frequency can be retrieved as follows:

  ```
  sudo modprobe msr

  # Minimum frequency
  echo "‘sudo rdmsr 0x620 --bitfield 14:8 --decimal‘00"

  # Maximum frequency
  echo "‘sudo rdmsr 0x620 --bitfield 6:0 --decimal‘00"
  ```

Alternatively, Likwid can be used to retrieve the CPU core and uncore frequencies as follows:

- the available CPU core frequencies (in GHz) can be retrieved as follows:

  ```
  likwid-setFrequencies -l
  ```

- the available CPU uncore frequency (including the minimum and maximum values) can be retrieved as follows:

  ```
  likwid-setFrequencies -p
  ```

Alternatively, x86_adapt can be used to retrieve the CPU uncore frequencies as follows:

```
# Minimum frequency (in 100 MHz)
x86a_read -n -i Intel_UNCORE_MIN_RATIO

# Maximum frequency (in 100 Mhz)
x86a_read -n -i Intel_UNCORE_MAX_RATIO
```

If likwid or x86_adapt are installed on the user system, then retrieving the CPU frequencies will not require sudo access.

D Examples

D.1 Modules on Taurus Cluster at TU Dresden

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.

D.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
```

D.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_bullxmpi1.2.8.4
```

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

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

D.2 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
```

On the Taurus cluster at TU Dresden, 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.

D.3 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 the Taurus cluster at TU Dresden in:

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

D.4 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, which are available on the Taurus cluster at TU Dresden:

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

D.5 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
      }
```

D.6 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.

On the Taurus cluster at TU Dresden, 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:

```
...
Significant regions are:

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


Significant region information
============================
Region name                  Min(t)          Max(t)       Time Dev.(%Reg) Ops/L3miss    Weight(%Phase)

void Comm::borders(Atom&)    0.001             0.001            2.6          109            0
void ForceLJ::compute_hal    0.013             0.014            2.9           97           68
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.

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 dynamism due to time deviation for miniMD.

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.

D.7 Tuning Model Creation

On the Taurus cluster at TU Dresden, 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.

```
#!/bin/sh

#SBATCH --time=5:00:00   # walltime
#SBATCH --nodes=2  # number of nodes requested; 1 for PTF and remaining for application run
#SBATCH --tasks-per-node=8 # number of processes per node for application run
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --partition=haswell
#SBATCH --mem-per-cpu=2500M   # memory per CPU core
#SBATCH -J "miniMD_PTF"   # job name
#SBATCH -A p_readex

echo "run PTF begin."

NP=8 # check against --ntasks and tasks-per-node

module purge
module use /projects/p_readex/modules
#module load readex/beta_gcc6.3
module load readex/ci_readex_bullxmpi1.2.8.4_gcc6.3.0

INPUT_FILE=in3.data #in.lj.miniMD
PHASE=INTEGRATE_RUN_LOOP

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib

export SCOREP_SUBSTRATE_PLUGINS=rrl
export SCOREP_RRL_PLUGINS=cpu_freq_plugin,uncore_freq_plugin
export SCOREP_RRL_VERBOSE="WARN"

module load scorep-hdeem/sync-xmpi-gcc6.3
export SCOREP_METRIC_PLUGINS=hdeem_sync_plugin
export SCOREP_METRIC_PLUGINS_SEP=";"
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

# Optionally, specify interval for HDEEM measurements
# All measurement triggerd by an event that happen in this interval will get the same energy reported
# This can help reduce the HDEEM overhead for measurements with a lot of region durations below 100ms
# However, it may lead to wrong results with PTF if the application consists of many short regions
export SCOREP_METRIC_HDEEM_SYNC_PLUGIN_GET_NEW_STATS=10

export SCOREP_MPI_ENABLE_GROUPS=ENV

psc_frontend --apprun="./miniMD_openmpi_ptf -i $INPUT_FILE"
             --mpinumprocs=$NP
             --ompnumthreads=1
             --phase=$PHASE
             --tune=readex_intraphase
             --config-file=readex_config.xml
             --force-localhost
             --info=7
             --selective-info=AutotuneAll,AutotunePlugins

echo "run PTF done."
```

To use the RAPL counter energy plugin replace the lines in the above script (that load the HDEEM module and set the Score-P metric plugins environment variables) with the following:

```
module load scorep_plugin_x86_energy
export SCOREP_METRIC_PLUGINS=x86_energy_sync_plugin
export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN=*/E
export SCOREP_METRIC_PLUGINS_SEP=";"
export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN_CONNECTION="INBAND"
export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN_VERBOSE="WARN"
export SCOREP_METRIC_X86_ENERGY_SYNC_PLUGIN_STATS_TIMEOUT_MS=1000
```

D.8 Production Run with Tuning Model

On the Taurus cluster at TU Dresden, a 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. This script is to be run from the location with the application's executable. The following script can be used to tune the application during its production run with RRL and compare the execution time and energy consumption with an untuned run of the application.

```
#!/bin/sh

#SBATCH --time=2:00:00
#SBATCH --nodes=1
#SBATCH --ntasks=8
#SBATCH --tasks-per-node=8
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --partition=haswell
#SBATCH --mem-per-cpu=2500M
#SBATCH -J "miniMD_rrl"
#SBATCH -A p_readex

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

energy_label="Energy"
rm -rf host_names.out
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 hostname >> host_names.out

#####
```

```
# application-specific setup here
INPUT_FILE=in3.data #in.lj.miniMD
#####

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib

############## start plain run ##############
export SCOREP_ENABLE_PROFILING="false"
export SCOREP_ENABLE_TRACING="false"
export SCOREP_SUBSTRATE_PLUGINS=""
export SCOREP_RRL_PLUGINS=""
export SCOREP_RRL_TMM_PATH=""
export SCOREP_MPI_ENABLE_GROUPS=ENV

# start measurements
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 clearHdeem
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 startHdeem
start_time=$(($(date +%s%N)/1000000))
# run untuned application
srun ./miniMD_openmpi_plain -i $INPUT_FILE
# stop measurements
stop_time=$(($(date +%s%N)/1000000))
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 stopHdeem
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 sleep 5
exec < host_names.out
while read host_name; do
  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 --nodelist=$host_name checkHdeem >> hdeem.out
done

# aggregate energy measurements from HDEEM
energy_total=0
if [ -e hdeem.out ]; then
  exec < hdeem.out
  while read max max_unit min min_unit average average_unit energy energy_unit; do
    if [ "$energy" == "$energy_label" ]; then
      read blade max_val min_val average_val energy_val
      energy_total=$(echo "${energy_total} + ${energy_val}" | bc)
    fi
  done
  time_total=$(echo "${stop_time} - ${start_time}" | bc)
  echo ""
  echo "Untuned run: Total time = $time_total ms, Total energy = $energy_total J"
  rm -rf hdeem.out
fi
############## end plain run ##############

############## start RRL-tuned run ##############
export SCOREP_ENABLE_PROFILING="false"
export SCOREP_ENABLE_TRACING="false"
export SCOREP_SUBSTRATE_PLUGINS="rrl"
export SCOREP_RRL_PLUGINS="cpu_freq_plugin,uncore_freq_plugin"
export SCOREP_RRL_TMM_PATH="tuning_model.json"
export SCOREP_MPI_ENABLE_GROUPS=ENV

# start measurements
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 clearHdeem
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 startHdeem
start_time=$(($(date +%s%N)/1000000))
# run RRL-tuned application
srun ./miniMD_openmpi_ptf -i $INPUT_FILE
# stop measurmenents
stop_time=$(($(date +%s%N)/1000000))
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 stopHdeem
srun -N 1 -n 1 --ntasks-per-node=1 -c 1 sleep 5
exec < host_names.out
while read host_name; do
  srun -N 1 -n 1 --ntasks-per-node=1 -c 1 --nodelist=$host_name checkHdeem >> hdeem.out
done

# aggregate energy measurements from HDEEM
```

```
energy_total=0
if [ -e hdeem.out ]; then
  exec < hdeem.out
  while read max max_unit min min_unit average average_unit energy energy_unit; do
    if [ "$energy" == "$energy_label" ]; then
      read blade max_val min_val average_val energy_val
      energy_total=$(echo "${energy_total} + ${energy_val}" | bc)
    fi
  done
  time_total=$(echo "${stop_time} - ${start_time}" | bc)
  echo ""
  echo "RRL-tuned run: Total time = $time_total ms, Total energy = $energy_total J"
  rm -rf hdeem.out
fi
############## end RRL-tuned run ##############
```