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

GA no. 671657

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

D3.2
Final mechanisms for run-time detection, switching, and calibration

Document type:	Report
Dissemination level:	Public
Work package:	WP3
Editor:	Per Gunnar Kjeldsberg (NTNU)
Contributing partners:	TUD, NTNU
Reviewers:	GNS, NUIG
Version:	1.0

Document history

Version	Date	Author/Editor	Description
0.1	03-Jan-2018	Per Gunnar Kjeldsberg (NTNU)	Inital structure version
0.2	29-Jan-2018	Per Gunnar Kjeldsberg (NTNU) Robert Schöne (TUD) Andreas Gocht (TUD) Umbreen Sabir Mian (TUD) Nico Reissmann (NTNU)	Draft for first internal review
0.3	19-Feb-2018	Per Gunnar Kjeldsberg (NTNU) Robert Schöne (TUD) Andreas Gocht (TUD) Umbreen Sabir Mian (TUD) Nico Reissmann (NTNU)	Draft for second internal review
1.0	28-Feb-2018	Per Gunnar Kjeldsberg (NTNU) Robert Schöne (TUD) Andreas Gocht (TUD) Umbreen Sabir Mian (TUD) Nico Reissmann (NTNU)	Final version for submission

Executive Summary

The objective of WP3 is to implement the READEX Run-time Library (RRL). Besides the RRL architecture implementation, this involves development of efficient scenario detection and switching mechanisms and the development of an efficient run-time scenario calibration mechanism. The final RRL architecture was presented in deliverable D3.1. Prototypes of scenario detection and switching, as well as concepts for runtime calibration, were presented in deliverable D4.2.

This deliverable describes the final mechanisms for runtime scenario detection and switching, and runtime scenario calibration.

When a production run of an application starts, the RRL is activated and the Application Tuning Model generated during Design Time Analysis is imported into the Tuning Model Manager. When a significant region in the application is entered, the composition of the current runtime situation is determined through interaction between Score-P and the different modules of the RRL. The runtime situation is used to determine the upcoming scenario, which again determines the setting of the different system configuration parameters such as core and uncore frequency. The actual switching of parameter settings is performed by Parameter Control Plugins.

If a runtime situation is detected that is not already in the Application Tuning Model, calibration is activated. A Machine Learning based method is used to quickly find an energy efficient system configuration for the new runtime situation. Performance Monitoring Counters are used to monitor hardware events that can be used to determine the configuration. When a configuration is found, the result is stored by the Tuning Model Manager for future use.

It is assumed that the reader of this document has already a good understanding of the READEX concepts from reading previous deliverables such as D4.1, D4.2, and D3.1.

Contents

1 Introduction

The objective of WP3 is to implement the READEX Run-time Library (RRL). Besides the RRL architecture implementation, this involves development of efficient scenario detection and switching mechanisms and the development of an efficient run-time scenario calibration mechanism. The final RRL architecture was presented in deliverable D3.1 [5]. Prototypes of scenario detection and switching, as well as concepts for runtime calibration, were presented in deliverable D4.2 [7].

This deliverable describes the final mechanisms for runtime scenario detection and switching, and runtime scenario calibration. These mechanisms all involve several modules in the RRL architecture depicted in Figure 1, as will be described in the following sections. Where detailed descriptions of mechanisms have already been presented in other deliverables, references are given to these while mainly an overview is given here.

The deliverable is structured as follows: Section 2 will present details about the implementation of the scenario detection mechanism. Section 3 presents details about the implementation of the scenario switching mechanism. In Section 4 we will present details about the implementation of the scenario calibration mechanism.

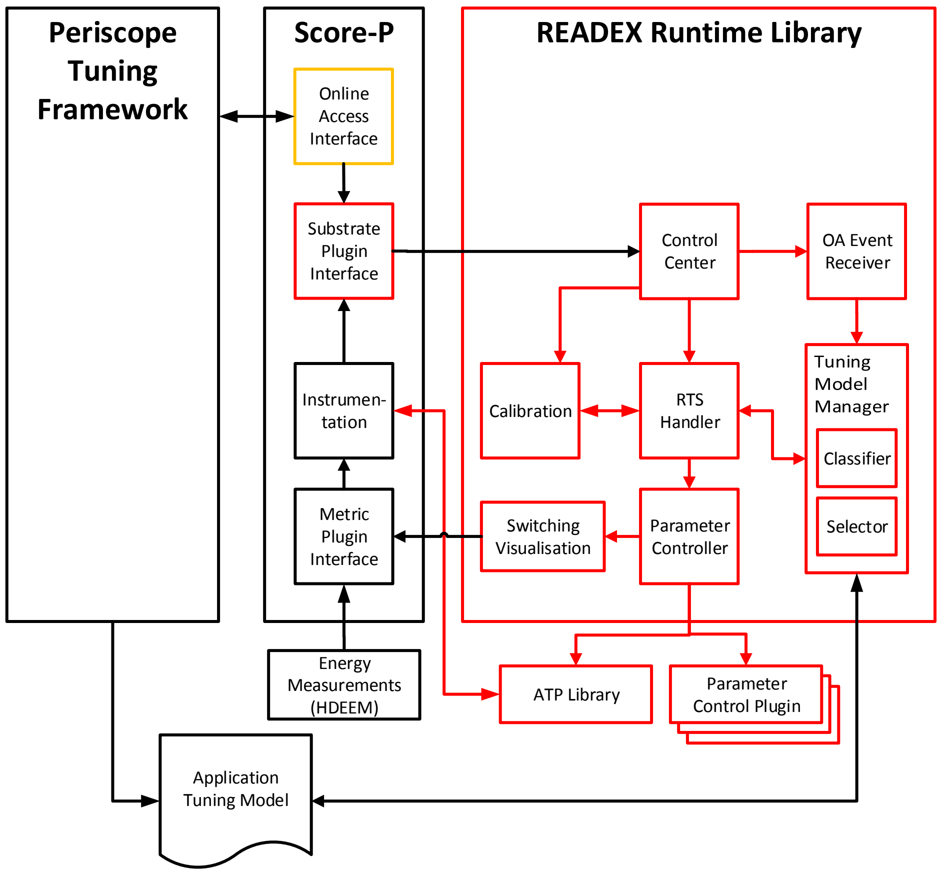

Figure 1: Overview of the READEX tool suite showing detailed RRL architecture, including interconnections with other parts of the READEX platform relevant for the content of this document.

2 Scenario Detection

Runtime detection of the upcoming scenario is a process that involves several modules in the RRL architecture depicted in Figure 1. The tasks of the different modules are described in the following subsections. Parts of the implementation have previously been presented in deliverables D3.1 [5, Section 2] and D4.2 [7, Section 2.3].

2.1 Score-P

The READEX project uses Score-P [13], which supports various mechanisms to instrument parallel applications. This includes

- compiler instrumentation, with support for numerous compilers for various processor architectures,
- instrumentation for parallelization mechanisms like MPI, OpenMP, CUDA, and OpenACC, and
- interfaces for manual instrumentation.

To apply these, Score-P has to be used as a compiler wrapper, e.g., by calling `scorep mpicc` instead of `mpicc` when compiling the program. Whenever an instrumented event is observed during the execution of the application, Score-P collects basic (timestamp, whether a region is either entered or exited, which region) and advanced information (e.g., various metrics like PAPI counters [1]) and calls the registered substrate to process this data. Traditionally, these substrates are profiling and tracing. In READEX, we extended this functionality to develop additional substrates as plugins via the Score-P Substrate Plugin Interface [15]. We use this interface to call the RRL.

2.2 Control Center

The Control Center acts as the basic interface between Score-P and the RRL. It instantiates the different subcomponents of the RRL, namely the RTS Handler, Tuning Model Manager (TMM), Online Access (OA) Event Receiver, and Calibration. The Control Center receives information about different events happening during an application run from Score-P and then decides which component gets which information.

The different interfaces between the Control Center and the other components of the RRL are shown in Figure 2. Upon approaching a new region, the Control Center receives notification from Score-P. The Control Center registers the new region with the TMM through a *register_region* call. The different region enter, region exit and user parameter events are forwarded to the Calibration module (interfaces not shown in Figure 2) and the RTS Handler through *enter_region*, *exit_region* and *user_parameter* calls, respectively. The RTS Handler also receives the create location and delete location events through *create_location* and

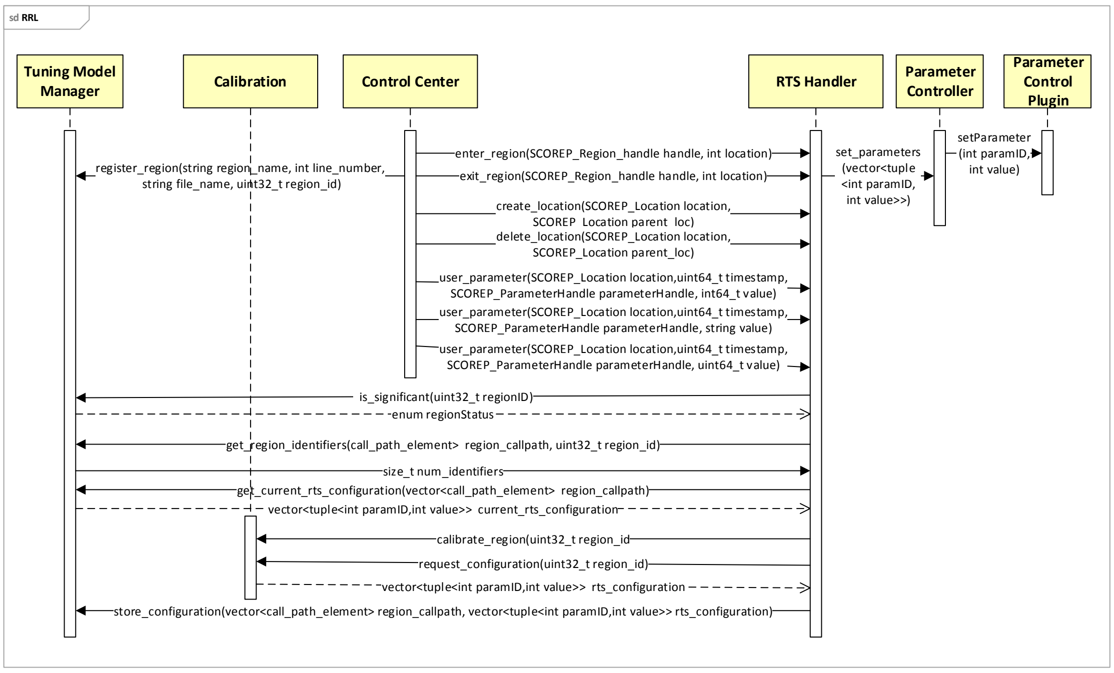
sd RRL
Tuning Model Manager
Calibration
Control Center
RTS Handler
Parameter Controller
Parameter Control Plugin
register_region(string region_name, int line_number, string file_name, uint32_t region_id)
enter_region(SCOREP_Region_handle handle, int location)
exit_region(SCOREP_Region_handle handle, int location)
create_location(SCOREP_Location location, SCOREP_Location parent_loc)
delete_location(SCOREP_Location location, SCOREP_Location parent_loc)
user_parameter(SCOREP_Location location,uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, int64_t value)
user_parameter(SCOREP_Location location,uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, string value)
user_parameter(SCOREP_Location location,uint64_t timestamp, SCOREP_ParameterHandle parameterHandle, uint64_t value)
set_parameters (vector<tuple <int paramID, int value>>)
setParameter (int paramID, int value)
is_significant(uint32_t regionID)
enum regionStatus
get_region_identifiers(call_path_element> region_callpath, uint32_t region_id)
size_t num_identifiers
get_current_rts_configuration(vector<call_path_element> region_callpath)
vector<tuple<int paramID,int value>> current_rts_configuration
calibrate_region(uint32_t region_id
request_configuration(uint32_t region_id)
vector<tuple<int paramID,int value>> rts_configuration
store_configuration(vector<call_path_element> region_callpath, vector<tuple<int paramID,int value>> rts_configuration)

Figure 2: Sequence diagram showing the interfaces between components of RRL used for scenario detection and scenario switching.

delete_location calls in case of OpenMP applications. The so called *Generic Commands* [5], which are generated by PTF through the OA Interface, are redirected to the OA Event Receiver. The details about the interfaces and working of OA Event Receiver are given in the Deliverable D3.1 [5].

2.3 RTS Handler

The RTS Handler maintains the current call stack and collects the additional identifiers. Additional identifiers are specified in the application by annotating them as user parameters, which are passed by Score-P to the RTS Handler via the Control Center as user parameter events. Upon receiving an enter region notification from Score-P 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 through the *is_significant* function. If it is an unknown region, the calibration mechanism is called and nothing else is done. A detailed description of the calibration mechanism is given in Section 4.

Table 1: Hash maps for storing ATM information.

Name	Key	Return value	Content
region	region_id	true / false	A list of all significant regions found at design time.
nidentifiers	region_id and call_path	number of identifiers	A list of call_path with region_id and the expected number of additional identifiers for each.
rts	set of identifier values	scenario_id	A list of rts's and the corresponding scenario identifier for each.
scenarios	scenario_id	parameter configurations	A list of scenarios and the corresponding configuration for all tuning parameters.

If the region is significant, then the RTS Handler passes the current call path to the TMM and asks the TMM for the number of additional identifiers through the *get_rts_identifiers* call. The TMM returns the number of additional identifiers that are expected. The RTS Handler collects these additional identifiers through user parameter events. Once the required additional identifiers are collected, the RTS Handler requests a new configuration from the Tuning Model Manager through the *get_current_rts_configuration* call and also passes the current call path to the Tuning Model Manager (TMM). The TMM returns back the configuration for the requested rts which is then passed to the Parameter Controller through the *set_parameters* call by RTS handler. The details about how exit region notifications are handled by the RTS Handler are given in Subection 3.2.

2.4 Tuning Model Manager

During Runtime Application Tuning (RAT), the Tuning Model Manager (TMM) is instantiated by the Control Center at the beginning of the application execution. The TMM then reads in and deserializes the Application Tuning Model (ATM) generated during Design Time Analysis (DTA), and stores the information as hash maps for efficient look up at runtime. Table 1 gives a description of the most important maps.

When the RTS Handler initiates a scenario detection process by asking if the current region is significant using the *is_significant* function, the TMM checks if this region is found in the ATM. The TMM returns information to the RTS Handler of whether the region was found or not. If the region was not found, runtime calibration is required. If the region was found, the RTS Handler next sends the current call path to the TMM. With this, the TMM can look up how many additional identifiers are expected for any rts related to this region and call path. Note that different call paths and regions may have different numbers of additional identifiers. For a given call path and region, however, the number of additional identifiers is always the same, even though the values of these identifiers may together constitute different rts's, and thus result in different scenarios being detected.

Finally, the TMM receives the complete rts, consisting of all the identifier values (region, call path, and additional identifiers). This is used as key to the hash map that stores the scenario id for each rts. Finding this scenario id completes the scenario detection process.

It may happen that the current region is present in the ATM, while the current call path, or subsequently, the current identifier values, are not. In this case, runtime calibration is required, and the TMM notifies the RTS Handler of this.

3 Scenario Switching

Runtime switching of the system configuration according to the detected scenario is a process that involves several modules in the RRL architecture depicted in Figure 1. The tasks of the different modules are described in the following subsections. Parts of the implementation have previously been presented in deliverables D3.1 [5, Section 2] and D4.2 [7, Section 2.3].

3.1 Tuning Model Manager

As described in Section 2, the TMM deserializes the ATM into hash maps (Table 1), one of which holds the system configuration for each scenario. The scenario detection process ends when the id for the upcoming scenario is found. This id is then used for hash map look up to find the system configuration that constitutes the scenario. This information is passed to the RTS Handler in response to its *get_current_rts_configuration* call.

A final task of the TMM is to store the configuration of rts's that have been calibrated at runtime. Upon receiving a new rts and its configuration from the RTS Handler (see Section 3.2 for details), a new entry in the hash map that stores system configurations is generated. The id for this entry is used to make a new entry also in the hash map storing rts's. Here, the identifier values of the new rts are used as key to select this id value. Finally, if the calibrated rts was coming from a previously unseen region, the region is also added to the hash map listing the significant regions.

3.2 RTS Handler

During scenario detection, if the region entered is detected to be significant by the RTS Handler, the RTS Handler gets a new configuration from the Tuning Model Manager for the generated rts (see Section 2 for details). This configuration is passed to the Parameter Controller, which sets the configuration through the respective Parameter Control Plugins (PCP) by calling the *set_parameter* function of the PCP.

Upon receiving an exit region event, the RTS Hander checks if the current region was set up for calibration. If yes, it requests the configuration for the currently exited region from the Calibration module by calling the *request_configuration* function. Once the RTS handler gets back the configuration from the calibration module, it passes this configuration to the TMM through the *store_configuration* call. TMM stores the new configfuration for the respective rts. If the region was not set up for calibration, then the Parameter Controller is informed that it might want to unset the current configuration. However, the actual decision whether or not the configuration gets unset is left to the Parameter Controller.

3.3 Parameter Controller

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

If the *no-reset* mode is selected, the current configuration stays active until a new configuration is set. The unset is ignored. The behavior of the no-reset mode is configurable using an environment variable named `SCOREP_RRL_CHECK_IF_RESET`. The no-reset mode is managed by a so called *configuration manager*, which has two different implementations each for the two possible values of the environment variable `SCOREP_RRL_CHECK_IF_RESET`, which are "no-reset" or "reset". By default, "reset" mode is enabled. The implementation of the configuration manager that manages the "reset" mode, saves the previous configurations on stack and changes the current configuration back to the previous configuration from the top of the stack on the exit of a region. The other implementation, which manages the "no-reset" mode, does not keep track of the previous configurations on stack as it leaves the current configuration set until a next configuration arrives.

3.4 Parameter Control Plugin

The PCPs perform the configuration of different hardware and system software resources.

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

Currently the following plugins are available:

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

A detailed description of the different PCPs can be found in Deliverable D1.1 [6].

3.5 Switching Visualization

Figure 1 shows the architecture of the READEX tool suite with the switching visualization module. This module is implemented as a metric plugin. It uses the metric plugin interface provided in Score-P to add the tuning parameters as metrics in Vampir [2, 3] traces and to get the tuning parameter values from RRL. The user can select these metrics in Vampir and visualize the switching pattern for each metric.

The metric plugin has to be loaded using the Score-P environment variable `SCOREP_METRIC_PLUGINS` and the user can specify if all of the tuning parameters, or only selected ones, need to be added to the Vampir trace. The tuning parameters that the user wants to visualize can be specified using the Score-P environment variable `SCOREP_METRIC_SCOREP_SUBSTRATE_RRL`. Any of the hardware, software and application tuning parameters can be chosen for visualization.

Figure 3 illustrates the switching of the CPU frequency and uncore frequency performed by RRL while tuning CPU frequency "CPU_FREQ" and uncore frequency "UNCORE_FREQ" for the Kripke benchmark. As can be seen in the Figure 3, the Vampir plots show the value of the tuning parameters for each region.

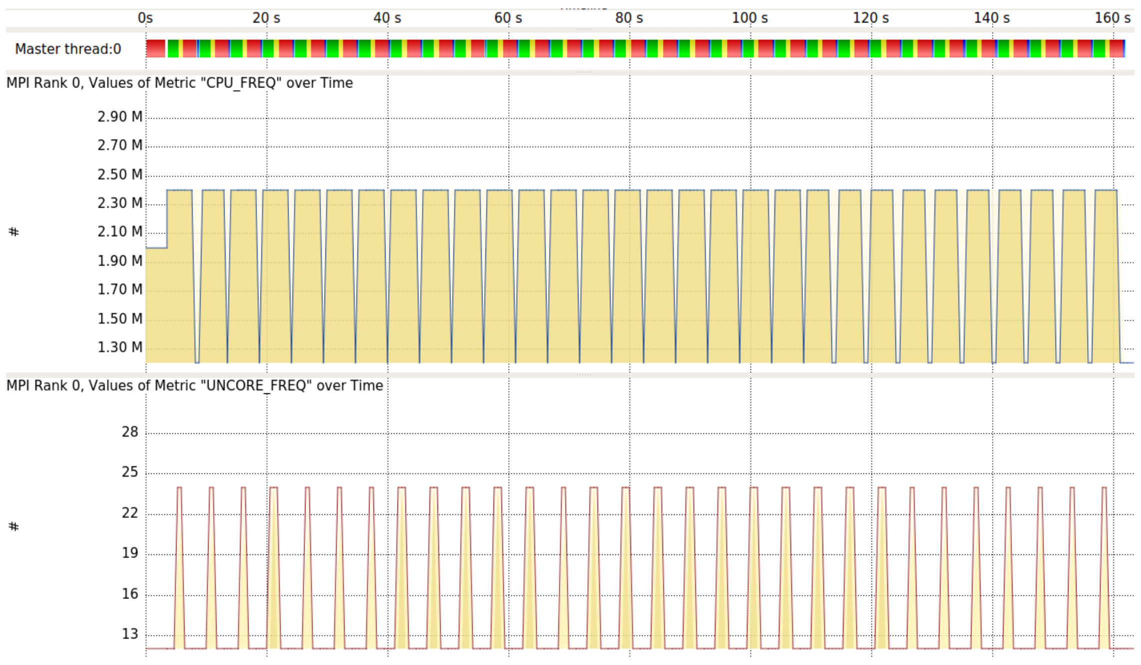

Figure 3: CPU_FREQ and UNCORE_FREQ switchings by RRL during Kripke run-time tuning

Further details about how to use the switching visualization metric plugin are given in Deliverable D4.3 [11].

4 Scenario Calibration

During RAT, we differentiate between known and unknown runtime situations (rts's). Known rts's describe those which have been encountered during DTA. So the optimal configuration for the rts's is known. However, unknown rts's describe those which have not been encountered during DTA. This might happen because some additional identifiers change, or simply because the application took an alternative code path. The goal of the calibration is to handle these unknown rts's during RAT.

The challenge lies in finding a good configuration in a short time. Searching for the optimal configuration as done during DTA is not feasible as it would significantly degrade the performance of the application. To avoid this we use a Machine Learning based method to determine a good configuration for an unseen rts. Using this method we can split the calibration in a training part and a detecting part. The training is done once per HPC system as described below. During RAT the trained model is used to detect a good configuration as described in Section 4.2. Once a configuration is found, it is stored in the TMM.

In difference to the DTA, we try to find a good configuration only in terms of core and uncore frequency. Adding the search for an optimal number of OpenMP threads would further increase the already large space of possible optimal configurations during training.

4.1 Training

Each Machine Learning algorithm needs a data basis, also called *feature vector*, to learn from. For supervised learning, an *optimization criterion* and a *target vector* are needed as well. In our case, the optimization criterion is to reduce the energy consumption of certain program functions. To do so, we change the frequency of the processor core and uncore, which represents the target vector. The training examples are generated by different energy optimal frequencies for monitored program functions. As feature vector, we use the hardware performance measurement counters (PMCs) as the data basis to learn from and predict a good configuration.

PMCs are CPU registers that can be used to count different hardware events, like executed instructions, cache accesses, and branch predictions. Each counter counts either a pre-defined event or an event that is specified in the accompanying control register. Within the control register, the event name and an umask can be specified. While the former indicates a class of events, e.g., the number of cache lines written to L2, the latter describes the event more precisely, e.g., which cache coherence state the lines have to have to be counted. Modern processor architectures are equipped with counters that measure events related to the processing units and core-related caches, and counters that observe the behavior of shared components within the uncore [14]. A detailed description for core and uncore counters on the Intel Haswell-EP platform can be found in [4, Chapter 18] and [10], respectively.

The purpose of the learning algorithm is to use these counters to predict the most energy efficient configuration of the processor frequency for a specific region. However, as there is only a limited number of PMCs available to measure the different events, it is not possible to collect all events at the same time. Hence, we need to find the appropriate core and uncore events. This is described in Section 4.1.1. Afterwards, we use them to train our Neural Network, which is explained in Section 4.1.2.

4.1.1 Selecting Relevant Performance Events

Suitable hardware events are those that contribute to a decision about the optimal frequencies. However, to gain the most accurate information, we need to filter out redundant information. For example, each L1 cache miss can also be defined as a L2 cache access. Therefore, only one of them needs to be measured, if at all.

Yoo [16] proposes the *Correlation-based Feature Selection* from Hall [8] to choose events without redundant information. Hall proposed a *Merit* to select features that correlate with the target vector but not with each other:

$$Merit = \frac{k\overline{r_{cf}}}{\sqrt{k + k(k-1)\overline{r_{ff}}}} \tag{1}$$

r_{cf} ... Correlation between target vector (c) and feature vector (f)

$\overline{r_{cf}}$ ... average of r_{cf}

r_{ff} ... Correlation between feature (f) and feature vector (f)

$\overline{r_{ff}}$ ... average of r_{ff}

k ... Number of features

For continuous class data, like the core frequency, Hall recommends the Pearson correlation for r:

$$r_{xy} = \frac{\sum xy}{n\sigma_x\sigma_y} \tag{2}$$

x, y ... Input vectors

σ_x, σ_y ... Standard deviation of x and y

n ... Length of the vectors x and y

However, the Pearson correlation is a linear correlation coefficient, but performance events might not correlate linearly with the optimal frequency as the energy consumption does not exhibit a linear correlation with the frequency. To relax the linearity criteria, we use *Kendalls* τ [12] to calculate the correlation between the different feature vectors or the different feature vectors and the target vector. Kendalls τ returns values between -1 and 1, where 1 indicates similarity (e.g., both values are rising) and −1 dissimilarity (e.g., one value is rising, while the other one is falling). As we are interested in both, values that are similar and dissimilar, but not in values that are not correlated at all, we use

$$\begin{aligned} r_{x,y} &= |\tau(x, y)| \qquad (3)\\ x, y \ldots &\text{ Input vectors}\\ \tau(x, y) \ldots &\text{ Kendall's } \tau \text{ for } x \text{ and } y \end{aligned}$$

Using the *Merit* we are now able to select relevant performance events for the training of a learning algorithm. To generate the input vectors for the learning algorithm, we remeasure the relevant events to generate new feature vectors.

4.1.2 The Learning Algorithm

For the implementation of the learning algorithm we chose Neural Networks (NN). Neural networks try to mimic the human brain, where computational units often referred to as *neurons* combine input values to produce an output value. Usually, the computed output value may be channeled to other neurons, depending on the number of layers incorporated by the NN [9].

Our aim is to collect feature vectors for a given configuration of the processor frequency that result in the lowest energy consumption. Moreover, in the training step, this data will be normalized by the runtime of the region and used as input to the NN. The optimal processor frequency for each region will be used as output.

4.2 Detection

Once the NN is trained, we can use it to find a good configuration for different application during RAT. The RRL loads the NN during initialisation. If an unknown rts is detected during RAT, the calibration is invoked. The RRL collects the relevant performance events of this rts at a default frequency. Once the rts is finished, the performance events are divided by the duration of the rts and passed to the calibration, which feeds them into the NN. The resulting frequencies are then passed to the TMM and stored for another occurrence of this rts.

5 Summary

The previous sections described the final implementations of the scenario detection, scenario switching, and calibration mechanisms used during Runtime Application Tuning by the READEX tool suite. As shown, there is close interaction between Score-P and various modules in the READEX Runtime Library to enable dynamic adaptation of the system configuration in order to improve the energy efficiency of applications during production runs. The calibration mechanism gives READEX the ability to handle runtime situations that were not encountered during Design Time Analysis.

The mechanisms described in this deliverable are implemented as part of the Beta prototype of the READEX tool suite, as described in deliverable D4.3.

References

[1] Shirley Browne, Jack Dongarra, Nathan Garner, Kevin London, and Philip Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC)*. IEEE, 2000. DOI: 10.1109/SC.2000.10029.

[2] 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.

[3] 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.

[4] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3*, 2018. [Online; accessed 05-January-2018].

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

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

[7] 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.

[8] Mark A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[9] Simon O. Haykin. *Neural Networks and Learning Machines.* Pearson, third edition, 2009.

[10] Intel Corp. *Intel Xeon Processor E5 v3 Family Uncore Performance Monitoring Reference Manual*, September 2014. [Online; accessed 04-March-2017].

[11] Venkatesh Kannan. D4.3 READEX tool suite version 2. Technical report, ICHEC-NUIG, 2018.

[12] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2), 1938. DOI: 10.2307/2332226.

[13] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf.

Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In H. Brunst, M. Müller, W. E. Nagel, and M. M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91. Springer, Berlin, 2012.

[14] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th International Conference on Performance Engineering (ICPE'17)*, page accepted for publication, Aquila, Italy, 2017.

[15] Robert Schöne, Ronny Tschüter, Thomas Ilsche, Joseph Schuchart, Daniel Hackenberg, and Wolfgang E. Nagel. Extending the functionality of Score-P through plugins: Interfaces and use cases. In Christoph Niethammer, José Gracia, Tobias Hilbrich, Andreas Knüpfer, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2016: Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing, October 2016, Stuttgart, Germany.* Springer International Publishing, 2017. DOI: 10.1007/978-3-319-56702-0_4.

[16] Wucherl Yoo, Kevin Larson, Lee Baugh, Sangkyum Kim, and Roy H. Campbell. ADP: Automated diagnosis of performance pathologies using hardware events. *SIGMETRICS Perform. Eval. Rev.*, 40(1), June 2012. DOI: 10.1145/2318857.2254791.