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

GA no. 671657

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

D2.3
Final Computation of Configurations

Document type:	Report
Dissemination level:	Public
Work package:	WP2
Editor:	Michael Gerndt (TUM)
Contributing partners:	TUM, NTNU, IT4I
Reviewer:	Robert Schöne (TUD)
	Uldis Locans (Intel)
Version:	1.0

Document history

Version	Date	Author/Editor	Description
0.1	24/12/17	Michael Gerndt (TUM)	1^{st} draft
0.2	2/02/17	Michael Gerndt, Madhura Kumaraswamy, Nico Reissmann	Full version for 1^{st} review
0.2	10/02/17	Robert Schne, Uldis Locans	1^{st} review
0.3	18/02/17	Michael Gerndt, Madhura Kumaraswamy, Nico Reissmann	Draft for 2^{nd} review
1.0	24/02/17	Michael Gerndt, Madhura Kumaraswamy, Per Gunnar Kjeldsberg	Integrated suggestions of the 2^{nd} reviews.

Executive Summary

This deliverable presents the final implementation of the READEX Design Time Analysis, which precomputes the tuning model that is used during production runs to tune the energy efficiency of the application. It is based on the Periscope Tuning Framework, an automatic tuning environment, that explores different tuning parameter configurations via tuning plugins. For READEX, two new tuning plugins were developed, one for intra-phase and one for inter-phase dynamism. The tuning model determines best configurations for scenarios of runtime situations. Scenarios are sets of identifiable instances of program regions that have the same characteristics and, based on that, the same best tuning parameters configurations. Therefore, the tuning model abstracts from individual region instances which leads to a compressed representation of the Design Time Analysis results. Tuning models from application runs with different input configurations are merged according to their input identifiers into a general application tuning model.

Contents

1 Introduction

The READEX tuning methodology consists of two steps: Design Time Analysis (DTA) and Runtime Application Tuning (RAT). DTA creates a tuning model that is used during production runs by the READEX Runtime Library (RRL) to perform RAT. READEX exploits application dynamism to reduce the application's energy consumption beyond static tuning. While in static tuning an optimized system configuration is enforced at program start and is not changed during execution of the application, dynamic tuning switches the configuration dynamically and can therefore exploit dynamic variation of application characteristics.

As outlined in Deliverable D4.1, READEX is able to exploit intra-phase and inter-phase application dynamism. *Intra-phase dynamism* identifies different characteristics of instances of program regions. These instances are called *runtime situations (rts)* (see Deliverable D4.1 for details).

Inter-phase dynamism identifies dynamism due to changing application characteristics over iterations of the main progress loop of the application. Each iteration of the progress loop typically simulates one time step and is called a *phase*. The loop body, the *phase region*, has to be specified by the application expert prior to DTA via the *Domain-Knowledge Specification Interface (DKSI)* (Deliverable D4.5). For example, variation in the characteristics of the application's phases results from the progress of the tool applied to the metal plate in the metal forming process simulated in INDEED [1] .

DTA is implemented by two new tuning plugins of the *Periscope Tuning Framework (PTF)*. A *tuning plugin* [6] captures expert knowledge about the tuning approach for a certain tuning aspect. It searches the multi-dimensional space of *system configurations*, where each of the dimensions is a *tuning parameter*. The approach for exploring the search space is determined by the plugin developer based on expert knowledge. Each tuning plugin consists of one or more *tuning steps*. In each tuning step, a *search algorithm* is used to determine a set of system configurations that are evaluated in experiments. PTF offers a number of search algorithms to the plugin developer. Each *experiment* is the execution of a phase with the tuning parameters set appropriately. At the end of the search, the *Application Tuning Model (ATM)* is generated. DTA is executed for a certain application input. This input can be described by *input identifiers*, i.e., key-value pairs. The input identifiers are then attached to the ATM. An external tuning model merger (Section 4) can be used to merge tuning models of different inputs together into one general ATM.

`readex-dyn-detect` analyzes the application for available intra-phase and inter-phase dynamism. The user can then select the `readex_intraphase` tuning plugin to exploit intra-phase dynamism, or the `readex_interphase` tuning plugin to exploit inter-phase and intra-phase dynamism. Since the two tuning plugins use different approaches, it is recommended to apply the `readex_intraphase` tuning plugin if there is no inter-phase dynamism.

The approach taken in PTF for DTA is focusing on the fast generation of high quality tuning models. DTA requires typically only one or two program runs to generate the model. In case of intra-phase dynamism, a single - frequently only partial - run of the application is sufficient to determine and evaluate the tuning model. For inter-phase dynamism three application runs are required for the analysis, the default configuration, and one to evaluate the tuning model quality.

The short tuning times result from three properties of the tuning plugins:

1. The experiments are executed for a single phase.
2. Advanced search algorithms require less experiments than exhaustive search.
3. Rts's are tuned independently.

These properties allow to perform DTA much faster than, for example, with the manual tuning approach based on MERIC [2]. The resulting tuning models result nevertheless in comparable tuning results.

The document explains the final design of the `readex_intraphase` plugin implementing tuning for intra-phase dynamism in Section 2 and the `readex_interphase` plugin tuning for inter-phase dynamism in Section 3. Section 4 describes advanced methods for tuning model generation as well as tuning model merging. The contents is summarized in Section 5.

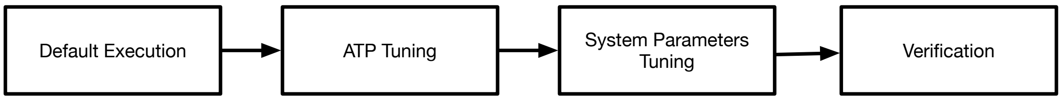


Figure 1: The `readex_intraphase` plugin executes four tuning steps: Default Execution, ATP Tuning, System Parameter Tuning, and Verification.

2 Intra-phase Tuning

PTF provides the `readex_intraphase` plugin dedicated to tuning based on intra-phase dynamism. In contrast to applications with inter-phase dynamism, the characteristics of the computation do not change over time, i.e., the sequence of phases.

It supports tuning for different objective functions, i.e., absolute energy consumption, energy delay product limiting the performance reduction due to energy tuning, and total cost of ownership are offered. The total cost of ownership objective function determines the overall costs of a job as being the sum of the energy costs plus the execution time dependent fraction of the HPC system costs, i.e., hardware and software investment as well as maintenance costs and personnel.

For all these objectives also a normalized version is provided where the objective value is normalized by a metric that linearly correlates to the amount of work, such as the number of AVX instructions. These normalized objectives allow to apply the `readex_intraphase` plugin also to codes that have a varying phase execution time but no varying characteristics, such as the computational intensity. Thus, the plugin tunes phases with different amount of work but of the same kind, such as more iterations of an iterative solver. The normalized objective would allow to compare the objective values of phases with different execution time.

The supported tuning parameters are application tuning parameters, core and uncore frequency, and number of threads. *Application Tuning Parameters (ATP)* are described in detail is Deliverable D4.5.

The `readex_intraphase` plugin is configured via the *READEX configuration file*. It is generated by `readex-dyn-detect` and specifies the objective function, the search algorithms to use, as well as the configuration for the system-level tuning parameters. For the tuning parameters, the possible values can be specified. Other configurations are related to technical details and are not important here.

Figure 1 illustrates the steps of the `readex_intraphase` plugin:

1. **Default Execution:** This step determines the objective value for the default execution, i.e., the execution with the default parameter settings provided by the batch system for the system parameters and the ATP specification in the code for ATPs. The objective value is used for determining the tuning result in the last step of the plugin.

2. **ATP Tuning:** The `readex_intraphase` plugin tunes ATPs first, since these select typically algorithmic alternatives and the selection is independent of the more fine-grained tuning provided by system parameters.

3. **System Parameters Tuning:** Once the algorithmic decisions were taken, the algorithms are tuned with the system parameters.

4. **Verification:** The verification step analyzes the results and determines the best system configuration for the phase region as well as the specific best settings for the significant regions. It then assesses the tuning result for this READEX configuration.

After the execution of the plugin, the tuning model is generated and, if necessary, merged with other already computed tuning models for different inputs by the external *Tuning Model Merger* described in Section 4.

2.1 Default Execution

The execution with the default tuning parameter settings is run as part of the initial gathering of the program's static information by the very first step of PTF after starting the application. PTF uses a specific analysis strategy to gather program regions and runtime situations. This strategy was extended to also gather the measurements required for the evaluation of the objective value, i.e., time and energy.

The default settings of the tuning parameters are given for the system parameters by the initialization of the batch job and by the default-value specification for the ATPs.

The analysis is run in PTF for the first phase of the application. The results are stored for comparison in the last plugin step.

2.2 ATP Tuning

This step investigates optimal settings for the given ATPs. ATPs allow the user to provide application specific tuning knobs based on expert application knowledge. The specification constructs are introduced in detail in Deliverable D4.5.

ATPs are implemented via the *ATP library*. It provides functions to specify their set of potential values and a default value. The set of values can be given by an interval or by enumeration. Furthermore, ATPs in the same *ATP domain* - a concept introduced in DKSI for structuring ATPs into sets of independent tuning parameters - can be connected by constraints that determine valid multi-dimensional points. The constraints are also specified by API calls.

In the first application phase, the ATP library generates an ATP configuration file that specifies the ATPs with their domain and possibly given constraints (Deliverable D4.5). PTF uses two new search strategies, `exhaustive_atp` and `individual_atp`, to determine optimal settings for the given ATPs based on this file.

The exhaustive strategy for ATPs explores all valid points in the given search space. The search space can consist of a single or multiple ATP domains. The search strategy first determines the valid points for each of the given ATP domains. It contacts the ATP server that is started during the initialization of the tuning plugin. The ATP server reads the ATP configuration file and answers queries of PTF about ATP domains, tuning parameters in the domains, and valid points of these ATPs. It employs the Omega Calculator [9] to solve the system of constraints that might have been specified for a given ATP domain.

Once the exhaustive strategy received the valid points for the domain, the set of system configurations is build as the crossproduct of those points.

The individual strategy for ATPs does not explore the crossproduct of the valid points of the domains but tunes the domains individually. It starts with the first domain and evaluates all valid points for this domain. It then fixes the best point for this domain by restricting the set of values to this single point and explores the next ATP domain until all domains were handled. The individual strategy uses the `exhaustive_ATP` strategy as a substrategy to explore the valid points of a domain.

The applied search strategy can be selected in the configuration file. The result of the first tuning step is an optimal configuration for the application tuning parameters.

2.3 System Parameters Tuning

The third tuning step explores the system-level tuning parameters. In this step, the optimal configuration for the ATPs obtained in the previous tuning step is fixed and applied in this tuning step.

The search strategy in this step is given in the configuration file and can be any of the search strategies of PTF. It is recommended to use the individual strategy for tuning the different parameters independently, which reduces the search time considerably.

The ranges of the tuning parameters are given in the configuration file as well. The search strategy generates the system configurations to be assessed via experiments. In an experiment for a system configuration, PTF measures the required objectives for the phase and for the rts's.

At the end of this tuning step, the tuning result for all tested system configurations are evaluated. By evaluating the objective for the phase region, the static best system configuration is determined. Afterwards, the tuning results for each rts are evaluated to compute the best system configuration for each rts. The combination of the best static system configuration and the rts-specific best configurations represents the READEX tuning model.

2.4 Verification

In the last step, the tuning plugin runs three experiments with the combined READEX configuration. To do so, PTF configures the RRL at the start of the phase with the phase system configuration and the rts-specific system configurations. During the next phase, the

RRL enforces the static configuration and dynamically switches system configurations for the rts's. The experiment is repeated three times to check for any variation in the results due to hardware variations or phase differences.

Finally, the plugin determines three values characterizing the tuning result. These are given below with the following notation:

$o_{phase}^{default}$: objective value of the phase under the default configuration

o_{phase}^{static} : objective value of the phase under the static best configuration

$o_{rts}^{default}$: objective value of the rts under the default configuration

o_{rts}^{static} : objective value of the rts under the static best configuration

o_{rts}^{opt} : objective value of the rts under the rts-specific optimal configuration

Please note that $o_{phase}^{opt} = o_{phase}^{static}$.

Static savings for the rts's: This is the accumulated savings for the static best configuration of the rts's compared to the default.

$$S_{RTS}^{static} = \sum_{rts \in RTS} (o_{rts}^{default} - o_{rts}^{static}) \;\; / \sum_{rts \in RTS} o_{rts}^{default} * 100$$

Dynamic savings for the rts's: This is the accumulated savings for the rts's with their specific best configuration compared to the value of the static best configuration.

$$S_{RTS}^{dynamic} = \sum_{rts \in RTS} (o_{rts}^{static} - o_{rts}^{opt}) \;\; / \sum_{rts \in RTS} o_{rts}^{static} * 100$$

Static savings for the whole phase: This is the comparison of the objective value for the best static configuration without rts specific configurations and the value for the default configuration.

$$S_{RTS}^{static} = (o_{phase}^{default} - o_{phase}^{opt}) \,/\, o_{phase}^{default} * 100$$

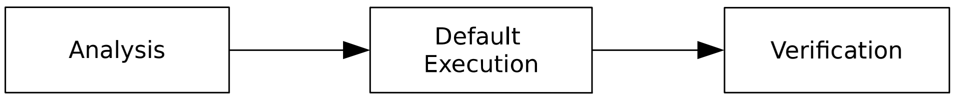


Figure 2: The `readex_interphase` plugin executes three tuning steps: Analysis, default execution and verification.

3 Inter-phase Tuning

The `readex_interphase` plugin is dedicated to tuning based on inter-phase dynamism for applications whose characteristics change across the sequence of phases.

The `readex_interphase` tuning plugin first groups phases that have similar characteristics or behavior into clusters and determines the best configuration for each cluster. It then determines the best configuration for the rts's of each created cluster.

During initialization, the plugin first reads and initializes the ranges of the tuning parameters from the READEX configuration file. It then initializes the `random` search strategy. This is done in order to reduce the search space of the tuning parameters. The plugin then reads the objective to tune the application for. If no objective is specified, the default objective is energy consumption.

Figure 2 illustrates the steps of the `readex_interphase` plugin:

1. **Analysis:** The analysis step uses the random search strategy [6] to create a number of experiments to request measurements for randomly selected configurations. It then clusters the phases based on the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [4] algorithm using normalized features (clustering aspects). Then, cluster-best configurations are determined for the phase as well as the rts's of the significant regions.

2. **Default Execution:** This step creates the same number of experiments as in the analysis step. Each experiment determines the objective value for the default execution, i.e., the execution with the default parameter settings provided by the batch system for the system tuning parameters. The objective values obtained for each phase as well as the rts's are then used for computing the savings at the end of the tuning plugin.

3. **Verification:** The verification step creates the same number of experiments as in the analysis step. This step determines if the theoretical savings computed in the analysis step match the actual savings incurred after switching the configurations.

Finally, the tuning model is generated after the end of the plugin.

3.1 Analysis

Before the start of the first step, PTF restarts the application, in order to start the analysis from the first phase. In the first step, the `random` search strategy is applied to collect the effect of different system configurations on the different phases. For each phase, a random system configuration is selected based on uniform distribution [6]. The number of experiments is determined by the `samples` specification in the READEX configuration file. In each experiment the objective value is gathered for the phase and the individual rts's of the significant regions. In addition, PAPI hardware metrics, such as the number of AVX instructions, L3 cache misses, and the number of conditional branch instructions are collected, which are later used to derive phase features.

Phase features, such as arithmetic intensity, capture characteristics of phases. Phases that have similar characteristics can be grouped together into a cluster. The mapping of phases into clusters based on the features enables PTF to select different best configurations for different phase clusters and also for the rts's of significant regions in different phase clusters.

The features should be chosen carefully as they have a high impact in selecting the best configuration for the phases in a cluster. Since the dynamism in many applications arises from the variation in the compute intensity and the control flow of the execution, compute intensity and the number of conditional branch instructions were chosen as the features for clustering. Compute intensity is determined by $\frac{\#AVX\ Instructions}{\#L3\ Cache\ Misses}$.

After executing the experiments, the analysis step normalizes the phase features and the objective values for all the phases and the rts's by, for example, the number of AVX instructions. This process is done for the same reason as described in Section 2. Then, the phase features (compute intensity and conditional branch instructions) are normalized using the *min-max* method. The min-max normalization method transforms the numeric range of a feature to a scale between between 0 and 1, as shown in the following formula:

$$x' = \frac{x - min(x)}{max(x) - min(x)}$$

3.1.1 Clustering

After the phase features are normalized, clustering is performed by the DBSCAN algorithm. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [4] is a density based clustering algorithm, which groups points that are closely packed together and have many nearby neighbors, resulting in high density regions. It marks points that do not have nearby neighbors and lie in low-density regions as noise. The algorithm requires two parameters to cluster the data points:

1. **minPts:** *minPts* determines the minimum number of points that must lie in the neighborhood to define a cluster. The *minPts* parameter was chosen to be 4 [10], which means that a neighborhood should have a minimum of 4 data points to be defined as a cluster.

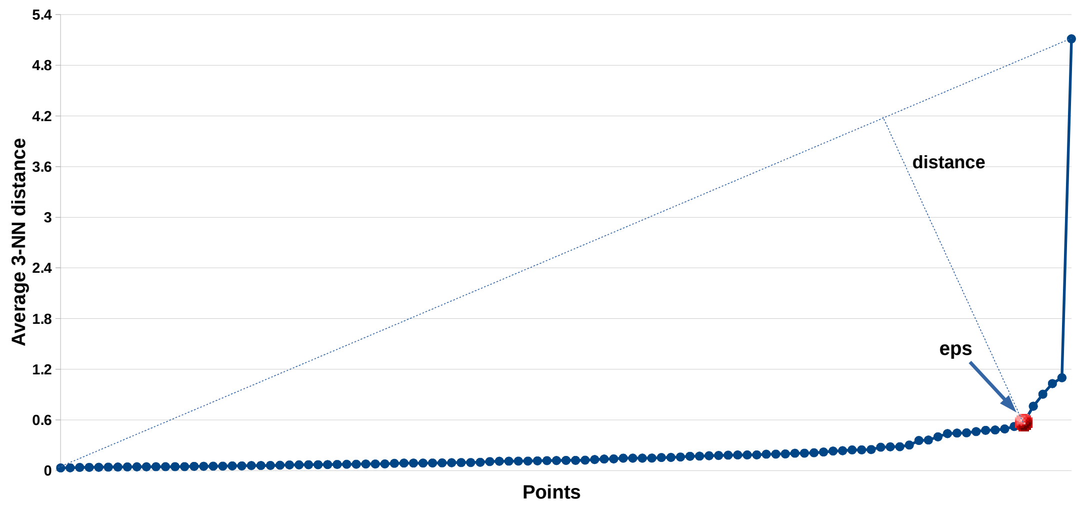


Figure 3: The *eps* parameter for DBSCAN is determined for the point that has the greatest distance from the line formed by the first and last point of the curve.

2. **eps:** Epsilon (*eps*) is the maximum distance between any two data points for them to be considered to be in the same neighborhood. *eps* is automatically determined using the *elbow* method [5]. First, this plugin computes the average 3-NN (3-Nearest Neighbor) Euclidean distances for all the data points. The distances are then arranged in the ascending order and a curve is drawn, as shown in Figure 3. The elbow is a sharp change in the average 3-NN distance plot. It is computed as the average 3-NN distance of the point that has the maximum distance to the line formed by the points with the minimum and the maximum 3-NN distance (the first and the last point on the curve).

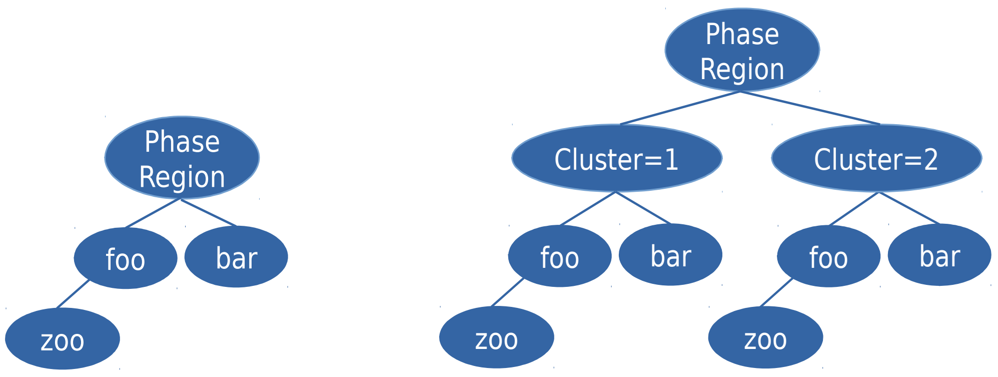
The phases are then clustered based on the *minPts* and *eps* parameters.

3.1.2 Computation of cluster-best configurations

After clustering the phases, the plugin selects the best configuration for each cluster based on the normalized objective value. This cluster-best configuration is then applied for all the phases of a particular cluster during the next application run. This step also determines the cluster-best configurations for the rts's of the significant regions. This information is then stored at the end of the Analysis tuning step.

3.2 Default Execution

In this tuning step, the application is restarted and the same number of experiments are executed with the default configuration. This step is performed to gather the measurements for the phase and the rts's of the significant regions in order to compute the savings.



(a) Before cloning. (b) After cloning.

Figure 4: Calling Context Graph (CCG) for a test application consisting of three regions foo, bar and zoo called in the phase region.

3.3 Verification Step

In the verification step, the `readex_interphase` plugin creates the same number of experiments as the previous tuning steps. Each experiment sets the configuration of the phase to its corresponding cluster-best configuration and the configuration of the rts's to their rts-specific best configuration in the phases' cluster.

If the phase was determined to be a noise point in the analysis step, it is executed using the default configuration of the batch system.

This step determines if the theoretical savings computed in the analysis step match the actual savings obtained after switching the configurations.

3.3.1 Cloning the Calling Context Graph (CCG)

Once all the experiments are completed, the plugin clones the children of the phase region of the Calling Context Graph (CCG) [1].

Figure 4a shows the initial CCG of the rts's of an example application containing regions foo, bar and zoo. Figure 4b depicts the case when all the call-tree nodes under the phase region are cloned, and new nodes are created for the n found phase clusters. Each of these cluster nodes contains information, including the phases of that cluster, and the ranges for the compute intensity and the conditional branch instructions. Each newly created cluster node represents the specific value of the new region identifier `Cluster`.

As for other region identifiers, the callpath of an rts starts with the phase region and includes the sequence of the region names and region identifiers that occur on the way to this region.

[1] A context sensitive version of a call graph.

In Figure 4a, the callpath for `foo` would be */PhaseRegion/foo*, while the valid rts's for `foo` after cloning would be */PhaseRegion/Cluster=1/foo* and */PhaseRegion/Cluster=2/foo.*

3.3.2 Computing the savings

The tuning results, including the objective values and the ranges of the phase features for each cluster as well as the objective values for the rts's of the significant regions per cluster are then inserted into the tree. Finally, the plugin determines the following three values to characterize the savings:

Static savings for the rts': This is the accumulated saving for the static best configuration of the rts's compared to the default over all the clusters.

$$S_{RTS}^{static} = \sum_{cluster=1}^{n} \sum_{rts \in RTS} (o_{rts,cluster}^{default} - o_{rts,cluster}^{static}) \ / \sum_{cluster=1}^{n} \sum_{rts \in RTS} o_{rts,cluster}^{default} * 100$$

Dynamic savings for the rts': This is the accumulated saving for the rts's with their specific best configuration compared to the value of the static best configuration over all the clusters.

$$S_{RTS}^{dynamic} = \sum_{cluster=1}^{n} \sum_{rts \in RTS} (o_{rts,cluster}^{static} - o_{rts,cluster}^{opt}) \ / \sum_{cluster=1}^{n} \sum_{rts \in RTS} o_{rts,cluster}^{static} * 100$$

Static savings for the whole phase: This is the accumulated saving for the the best static configuration of the phase without rts specific configurations compared to the default configuration over all the clusters.

$$S_{RTS}^{static} = \sum_{cluster=1}^{n} (o_{phase,cluster}^{default} - o_{phase,cluster}^{opt}) \ / \sum_{cluster=1}^{n} o_{phase,cluster}^{default} * 100$$

where,

$o_{phase,cluster}^{default}$	=	objective value of the phase under the default configuration for a particular cluster
$o_{phase,cluster}^{static}$	=	objective value of the phase under the static best configuration for a particular cluster
$o_{rts,cluster}^{default}$	=	objective value of the rts under the default configuration for a particular cluster

$o^{static}_{rts,cluster}$	=	objective value of the rts under the static best configuration for a particular cluster
$o^{opt}_{rts,cluster}$	=	objective value of the rts under the rts-specific optimal configuration for a particular cluster

Please note that $o^{opt}_{phase,cluster} = o^{static}_{phase,cluster}$.

3.4 Cluster Prediction at Runtime

For runtime tuning, the cluster number is added to the application as a region identifier. This is done by defining a Score-P user parameter (Deliverable D4.5) via `SCOREP_USER_PARAMETER(handle, "Cluster", predict_cluster())` for Fortran or `SCOREP_USER_PARAMETER("Cluster", predict_cluster())` for C/C++ applications immediately after the OA phase region annotation, `SCOREP_OA_PHASE_BEGIN()`, as shown in Listing 1. The user parameter definition sets the name to `Cluster`, and calls the *predict_cluster* function from an external cluster prediction library.

Listing 1: Score-P user parameter definition for cluster prediction during runtime.

```
...
SCOREP_OA_PHASE_BEGIN()
SCOREP_USER_PARAMETER_INT64("Cluster", predict_cluster())
...
SCOREP_OA_PHASE_END()
```

First, the predictor method requests from the Tuning Model Manager (TMM) the ranges of the phase features, and also the set of all phases belonging to the clusters determined in the analysis step of the `readex_interphase` plugin. The TMM returns the information for the first n phases that were already run during DTA.

The `predict_cluster` method initializes the PAPI library once at the beginning of the first phase. It then collects hardware metrics for the conditional branch instructions and the compute intensity, which is computed from the L3 cache misses and the AVX instructions. It performs the cluster prediction for the current phase similar to a one-bit branch prediction scheme. The branch prediction scheme predicts that the current branch will be taken if it was taken previously. Similarly, the cluster prediction function assigns the upcoming phase to the current cluster until the cluster is mispredicted, as shown in Figure 5.

The predictor method then assigns the value of the predicted cluster number, and communicates this to the RRL which looks up the configuration for the cluster number and sets it for the remainder of the phase. The configuration switching for the rts's called inside the current phase is then handled as usual by the RRL.

At the end of the current phase, a check is made to determine if there was a misprediction of the cluster. The collected PAPI metrics for the compute intensity and the conditional branch instructions are checked against the ranges of the phase features for the predicted cluster. If

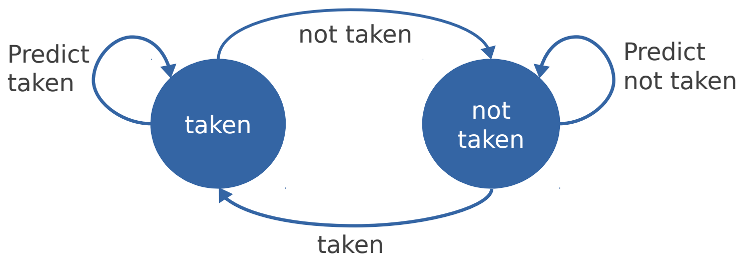


Figure 5: Runtime cluster prediction works similar to a one-bit branch prediction scheme with a 1-bit saturating counter, where a branch is predicted to be taken until it is mispredicted.

the values fall within the ranges for the predicted cluster, no action is taken. If the values fall within the ranges of another cluster, a correction is performed. If the values of the phase features do not fall in the ranges of any known cluster, the phase is assigned as a noise point, and the default configuration is set.

4 Tuning Model Generation

At the end of DTA, the tuning model is generated based on the best system configuration computed via the tuning plugins. The tuning model size is reduced by clustering rts's into scenarios. A best configuration is then given for a scenario and applied to all its rts's.

This section describes the advanced clustering of runtime situations into scenarios and the merging of tuning models created by PTF for different input identifiers.

4.1 Advanced Clustering of Runtime Situations

The task of the scenario identification module is to cluster rts's into scenarios. So far, the module could only cluster rts's with *identical* system configurations, but with the advanced scenario identification it is possible to cluster rts's with *similar* system configurations.

The advanced clustering is implemented using hierarchical clustering [7] and proceeds in three phases: dendrogram generation, cluster generation, and scenario creation. The first phase generates a *dendrogram*, *i.e.*, a tree illustrating the (dis-)similarity of rts's based on their system configurations, while the second phase uses this dendrogram to find a clustering for the rts's. The third phase uses these clusters to create the scenarios and their system configurations.

4.1.1 Dendrogram Generation

The dendrogram generation phase arranges all rts's into a dendrogram based on the similarity of their system configurations in the tuning parameter space. A dendrogram is a tree that expresses the (dis-)similarity of objects based on a distance metric. The tree's leaves represent the rts's, whereas all intermediate nodes represent the clusters of these rts's. The distance between merged clusters is expressed as the height of each intermediate node from its leaves. It is proportional to the intergroup (dis-)similarity of an intermediate node's children.

An example is shown in Figure 6 and 7. Figure 6 tabulates the system configurations for six different rts's, while Figure 7 shows these configurations graphically. The resulting dendrogram is shown in Figure 8a. The leaves of the dendrogram show the individual rts's and the intermediate nodes the potential clusters.

The algorithm treats every rts in the tuning parameter space as a cluster with only a single element, *i.e.*, the rts itself. It uses an inter-cluster distance metric to always merge the closest clusters, recomputes the distance for the newly created cluster to all other clusters, and proceeds to merge the next closest clusters. The algorithm continues until all clusters are merged into a single cluster. In the example in Figure 8a, the algorithm first merges rts5 and rts6 into a cluster, then rts1 and rts2, and then this cluster with rts3, and so forth.

We use Lance-Williams algorithm [8] to recursively compute the inter-cluster distance at every step of the algorithm in order to find the next pair of clusters. Suppose that clusters C_i

RTS	System Configuration	
	Core Freq.	Uncore Freq.
rts1	1200	1400
rts2	1300	1300
rts3	1200	1200
rts4	1700	1600
rts5	2100	2000
rts6	2300	2000

Figure 6: System Configurations for six rts'.

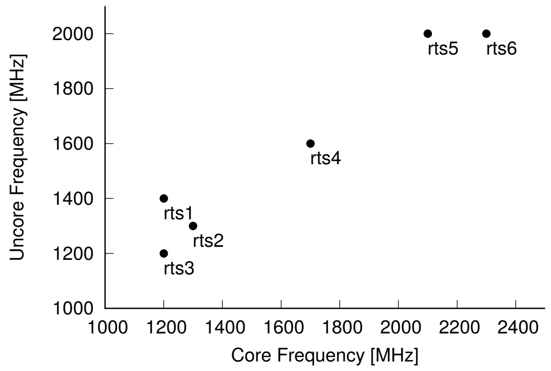


Figure 7: Graphical representation of rts system configurations.

Distance	α_i	α_j	β	γ
single-linkage	0.5	0.5	0.0	-0.5
complete-linkage	0.5	0.5	0.0	0.5
group-average	$\frac{n_i}{n_i+n_j}$	$\frac{n_j}{n_i+n_k}$	0.0	0.0
weighted group-average	0.5	0.5	0.0	0.0
centroid	$\frac{n_i}{n_i+n_j}$	$\frac{n_j}{n_i+n_k}$	$\frac{-n_i n_j}{(n_i+n_j)^2}$	0.0
Ward's method	$\frac{n_i+n_k}{n_i+n_j+n_k}$	$\frac{n_j+n_k}{n_i+n_j+n_k}$	$\frac{-n_k}{n_i+n_j+n_k}$	0.0

Table 1: Supported distance metrics.

and C_j are merged next, *i.e.*, $C_i \cup C_j$, then we can compute the distance to another cluster C_k as follows:

$$d_{(ij)k} = \alpha_i d_{ik} + \alpha_j d_{jk} + \beta d_{ij} + \gamma |d_{ik} - d_{jk}|$$

where d_{ij}, d_{ik}, and d_{jk} are the pairwise distances between cluster C_i, C_j, and C_k, and $d_{(ij)k}$ the distance between the new cluster $C_i \cup C_j$ and C_k. The parameters α_i, α_j, β, and γ are parameters that may depend on cluster size n, and can be used to express different inter-cluster distances. For example, instantiating α_i and α_j with 0.5, β with 0.0, and γ with -0.5 results in single-linkage clustering. Thus, Lance-Williams clustering algorithms are parametric in the inter-cluster distance. Table 1 gives an overview of the supported distance metrics.

4.1.2 Cluster Generation

The dendrogram produced by the previous phase enables us to create a clustering of the rts's by performing a tree cut. The cut of a tree is the partitioning of the tree's nodes into disjoint subsets. An example is shown in Figure 8b. The cut intersects with the dendrogram's edges

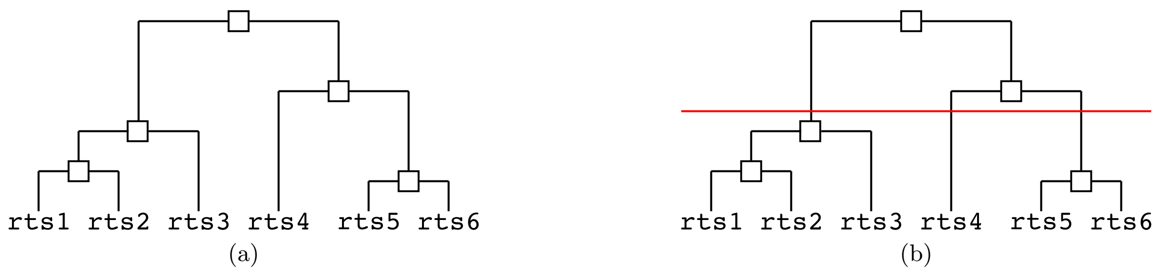


Figure 8: Dendrograms.

at three points and therefore partitions the tree into three disjoint subsets. These are the clusters $C_1 = \{rts1, rts2, rts3\}$, $C_2 = \{rts4\}$, and $C_3 = \{rts5, rts6\}$.

Thus, the point where we cut a dendrogram determines the number of resulting clusters. This point should be chosen such that it minimizes the dispersion of data within clusters (SS_w), while maximizing the dispersion between the clusters (SS_B). The overall dispersion of data within clusters for a given number of clusters k can be computed as follows:

$$SS_W = \sum_{i}^{k} \sum_{x \in C_i} ||x - m_i||^2 \tag{1}$$

where x is a data point, m_i the centroid of cluster i, and $||x - m_i||$ is the distance (*e.g.* Euclidean distance) between the data point and the centroid. SS_W decreases as the number of clusters increases as each cluster becomes smaller and tighter packed with a bigger k.

The dispersion between clusters for a given number of clusters k can be computed as follows:

$$SS_B = \sum_{i}^{k} n_i ||m_i - M||^2 \tag{2}$$

where n_i is the number of elements in C_i, M the centroid of all clusters, and $||m_i - M||$ the distance between the centroid of cluster C_i and the centroid of all clusters. SS_B measures the variance of all cluster centroids from the global centroid and a higher SS_B value means that clusters are more spread out.

We can combine both metrics for a given number of clusters k to compute the *Calinski-Harabasz Index* [3] as follows:

$$CHI = \frac{SS_B}{SS_W} \cdot \frac{N - k}{k - 1} \tag{3}$$

where N is the total number of data points (*i.e.* rts's). As SS_W keeps on decreasing with increasing k, the ratio $\frac{SS_B}{SS_W}$ is going to the biggest for the optimal number of clusters. Thus, we can compute the optimal number of clusters k_{best} as follows:

$$k_{best} = k \in 2, .., N \ where \ k \ maximizes \ CHI \tag{4}$$

where N is the total number of data points (*i.e.* rts's). The optimal number of clusters k_{best} enables us to compute the point where we cut the dendrogram and create the clusters.

4.1.3 Scenario Creation

The clusters can now be grouped into scenarios and an overall system configuration for the scenario can be computed. This system configuration is then used for all rts's in the cluster at runtime. Currently, two methods are supported to create a system configuration for a scenario. The first one simply picks a random configuration from among all the rts's, whereas the second one computes the average for each tuning parameter from all the system configurations of the rts's.

4.2 Merging of Tuning Models

The DTA is always executed for a certain application input. This input can be described by input identifiers, which are a list of key-value pairs. These input identifiers are attached to the ATM and therefore permit PTF to produce tuning models for specific inputs. Each of these tuning models contains the system configurations for the found rts's for a specific set of input identifiers. In order for all of the tuning information from the different ATMs to be usable by the RRL, these tuning models must be merged into a single tuning model. This is the task of the tuning model merger.

The tuning model merger is a standalone program that takes all the ATMs as input on its command line and outputs a new tuning model incorporating all the rts's from the input tuning models. The program does this by first deserializing all ATMs. It extracts all tuning information from the ATMs, such as rts's and their system configurations as well as the corresponding input identifiers. The scenarios from the individual ATMs are discarded. Next the tuning model merger filters all rts's in order to avoid duplicated rts's in the final tuning model. The next step is to produce a new set of scenarios by clustering all rts's as described in Section 4.1. Finally, the tuning model merger serializes the merged tuning model information and outputs the new ATM in the JSON format.

The merged ATM is read by the Tuning Model Manager (TMM) during Runtime Application Tuning (RAT). This is detailed in Deliverable D3.2. During RAT, the TMM handles the input identifiers the same way as any other identifier.

5 Summary

READEX provides Design Time Analysis for the computation of optimal system configurations. The final version of DTA supports intra-phase and inter-phase dynamism with two specialized PTF tuning plugins. The intra-phase tuning plugin supports ATPs as well as system-level tuning parameters. The inter-phase tuning plugin implements a tuning strategy that handles dynamism across phases via (1) experiments for randomly chosen system configuration and (2) clustering phases with similar characteristics. RAT will predict the cluster of the next phase based on the characteristics of previous phase and the optimal system configurations for the predicted cluster will be applied.

Tuning models obtained from DTA for different application input data sets are combined into a general tuning model. RAT will select system configurations from the tuning model according to input parameters given for a production run.

The phase-based implementation of DTA in PTF reduces the analysis time significantly. Instead of a full application run for each possible system configuration, the number of potentially partial application runs during DTA is independent of the size of the number of possible system configurations.

References

[1] INDEED – Highly Accurate Finite Element Simulation for Sheet Metal Forming. `http://gns-mbh.com/products/indeed`. Last accessed October 14, 2015.

[2] MERIC Tool for Energy Measurement - GIT Repository. https://code.it4i.cz/xvysoc01/meric.git.

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

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[5] Manisha Naik Gaonkar and Kedar Sawant. Autoepsdbscan: Dbscan with eps automatic for large dataset. *International Journal on Advanced Computer Theory and Engineering*, 2(2):11–16, 2013.

[6] Michael Gerndt, Eduardo César, and Siegfried Benkner, editors. *Automatic Tuning of HPC Applications - The Periscope Tuning Framework*. Shaker Verlag, Aachen, 2015.

[7] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

[8] G. N. Lance and W. T. Williams. A general theory of classificatory sorting strategies. *Computer Journal*, 9(4):373–380, 1967.

[9] Evan Rosser, Wayne Kelly, Bill Pugh, Dave Wonnacott, and Tatiana Shpeisman. The omega project.

[10] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data Min. Knowl. Discov.*, 2(2):169–194, June 1998.