GA no. 671657

D2.1

Analysis of tuning potential and scenario identification

Document type:	Report
Dissemination level:	Public
Work package:	WP2
Editor:	Per Gunnar Kjeldsberg (NTNU)
Contributing partners:	TUD, NTNU, TUM
Reviewer:	Kai Diethelm (GNS)
	Zakaria Bendifallah, Othman Bouizi (Intel)
Version:	1.2

Document history

Version	Date	Author/Editor	Description
0.1	16/06/16	Per Gunnar Kjeldsberg (NTNU)	1^{st} TOC draft
0.2	17/06/16	Per Gunnar Kjeldsberg (NTNU)	Added reviewers and writing responsibilities
0.3	20/06/16	Per Gunnar Kjeldsberg (NTNU)	Moved to tex-format
0.4	03/08/16	Anamika Chowdhury (TUM)	Initial version of tuning potential analysis
0.5	04/08/16	Mohammed Sourouri (NTNU) Per Gunnar Kjeldsberg (NTNU)	Initial version of scenario identification
0.6	06/08/16	Michael Gerndt (TUM)	Improved version of tuning potential analysis
0.7	08/08/16	Per Gunnar Kjeldsberg (NTNU)	Added Exec. Summary, Intro and Summary. Version for first internal review.
0.8	13/08/16	Michael Gerndt (TUM)	Update after review. Still some work to be done in the section on significant region detection.
0.9	21/08/16	Michael Gerndt (TUM)	Updated significant region detection.
1.0	21/08/16	Per Gunnar Kjeldsberg (NTNU)	Updated Summaries, Intro and Scenario identification. Version for second internal review.
1.1	26/08/16	Per Gunnar Kjeldsberg (NTNU)	Intermediate version partly answering comments from second review.
1.2	30/08/16	Michael Gerndt (TUM) Per Gunnar Kjeldsberg (NTNU)	Addressed all the final comments of the second review.

Executive Summary

This deliverable describes two different parts of the READEX tool development, the tuning potential analysis and the approach for scenario identification.

The tuning potential analysis is the very first step of the READEX Design Time Analysis (DTA). It analyses the application with respect to its suitability for the READEX tuning methodology. If insufficient tuning potential is found, READEX terminates advising the user to apply regular static tuning instead. Two automatic tools have been developed, `scorep-autofilter` and `readex-dyn-detect`. `scorep-autofilter` measures the granularity of program regions and generates a list of frequently executed regions with too fine granularity (too short execution time per instance). This list is used to filter out these regions from instrumentation, thus reducing the instrumentation overhead. `readex-dyn-detect` first selects significant regions from the remaining set of regions after filtering. These are regions coarse grained enough for reconfiguration overhead to be negligible, that are not nested in other significant regions, and that cover as much as possible of the overall execution time. Finally, `readex-dyn-detect` performs dynamism analysis of the significant regions. Variations in execution time for individual executions of a given significant region and in difference in compute intensity between significant regions signify dynamism that READEX can exploit. If this is found, READEX continues to the next steps. If no such READEX exploitable dynamism is detected, the tool terminates generates a message stating that the application should be executed without READEX.

The second part of this deliverable describes the final step of the DTA, the approach for scenario identification. In between the tuning potential analysis and the scenario identification, the Periscope Tuning Framework is used to search for the best possible system configuration for each runtime situation (rts). The result of this search is stored in a RTS database that includes information about the context element as well as the best found configuration of each rts. The database is iterated to group rts's with identical configurations in scenarios. This is followed by a grouping of rts's with similar configurations, using a similarity score that aggregates the closeness of a set of different system parameters. The clustering limits the number of scenarios and thus reduces the associated runtime overhead. When the set of scenarios has been generated, a scenario classifier is defined using a map data structure to determine the upcoming scenario at runtime. The identifier values are keyed to their respective scenario. Furthermore, a configuration selector is generated for each scenario for use at runtime to select the configuration of the chosen scenario. The scenario set, the classifier and the selectors are stored at design time in the Application Tuning Model as a serialised in-memory representation. It is then read and deserialised at runtime to enable scenario detection and configuration switching.

Contents

1 Introduction

This deliverable describes two different parts of the READEX tool development, the tuning potential analysis and the approach for scenario identification. These are the first and last steps in the READEX Design Time Analysis (DTA) stage. In between the tuning potential analysis and the scenario identification, the Periscope Tuning Framework (PTF) is used to search for the best possible system configuration for each runtime situation (rts).

While writing this document it has been assumed that readers are familiar with READEX deliverable D4.1 Concepts for the READEX Tool Suite [3]. Concepts defined in D4.1 are consequently not explained in detail here. Note, however, that the rts concept in general should be understood as the set of multiple executions of the same significant region with identical context elements (identifiers with values). In D4.1 these are described as individual rts's, while they here are linked together to achieve more efficient handling and compact data structures. See Section 3.1 for details.

This deliverable is organised as follows. Section 2 presents the tools developed for tuning potential analysis. The section is subdivided into descriptions of a tool for automatic reduction of instrumentation overhead, for significant region detection, and dynamism analysis. Section 3 describes the approach developed for scenario identification, mainly subdivided into the implementations of scenario clustering (grouping of rts's into Scenarios), scenario classifier generation, and configuration selector generation. This is followed by a summary in Section 4.

2 Tuning Potential Analysis

The first step of Design Time Analysis (DTA) (see Deliverable D4.1) is to analyse the application for its suitability for the READEX tuning methodology. If sufficient tuning potential exists, DTA determines a tuning model which is later on passed to the READEX Runtime Library (RRL). An application has tuning potential for the READEX methodology if it exposes intra- and/or inter-phase dynamism. Intra-phase dynamism results from variations in the characteristics of significant regions within a phase, while inter-phase dynamism results from variations in the execution characteristics of different program phases.

Two automatic tools were developed in READEX to analyse the application's dynamism and thus its tuning potential. The first tool, `scorep-autofilter`, analyses the overhead induced by Score-P's compiler instrumentation and generates a Score-P filter file that suppresses measurement overheads of selected program regions.

The second tool, `readex-dyn-detect`, analyses the instrumented application and determines regions suited for the READEX methodology due to their granularity. These regions are called *significant regions*. In addition, it analyses the intra- and inter-phase tuning potential, i.e., the dynamism in the application that can be exploited by runtime tuning.

Section 2.1 presents the automatic computation of the filter file, Section 2.2 outlines the detection of significant regions and Section 2.3 discusses the identification of the application's tuning potential.

2.1 Automatic reduction of instrumentation overhead

The first step of DTA is to reduce the instrumentation overhead induced by Score-P. This is due to automatic instrumentation by the compiler. For too fine granular regions the overhead might dominate the execution of the region. The Score-P monitoring system can expunge a list of regions from measurement. This list of fine granular regions is manually created by the user. This means that the user has to explore the frequency and execution time of all regions. Based on these, one can create the filter file with the names of too fine granular program regions.

For the READEX DTA we implemented an automatic tool to free the user from manually determining the filter file. The `scorep-autofilter` tool delves into the measurements for the program regions and generates a list of frequently executed, too fine granular regions to reduce the instrumentation overhead.

Figure 1 shows the whole workflow. The first step is to instrument the targeted application with the Score-P measurement infrastructure. The instrumenter inserts probes around program regions that trigger measurements at the enter and exit events. This instrumentation can be performed automatically by the compiler or manually by the user. Score-P is able to instrument different region types, such as program functions, MPI calls, and OpenMP parallel regions.

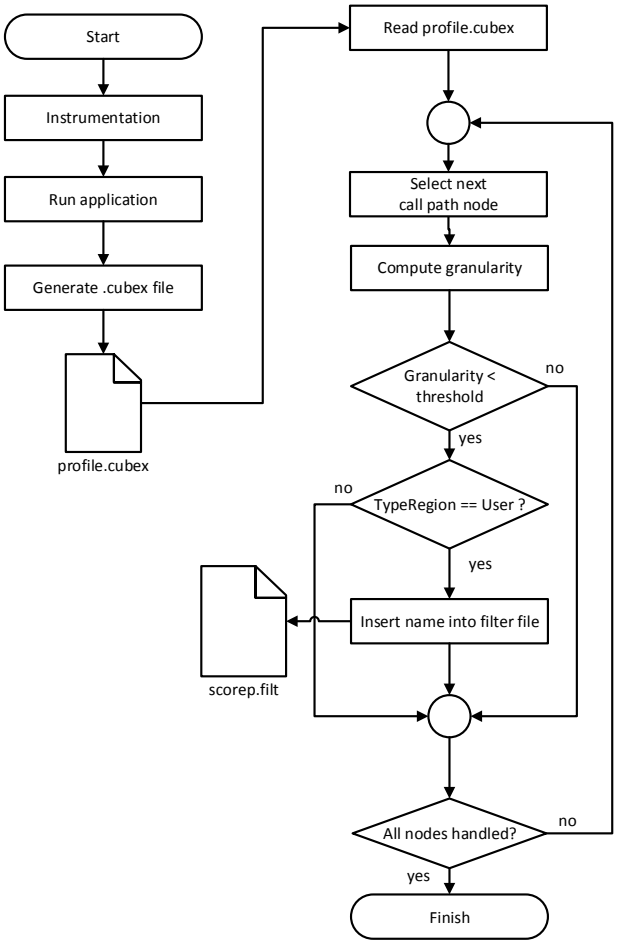

Figure 1: Workflow for creating the Score-P filter file with `scorep-autofilter`.

The `scorep-autofilter` tool analyses the performance data in the file `profile.cubex`, which is generated by Score-P and stores the data in a high level data model, the *cube* format [4]. This data model is represented by a tree of the region's call paths. Each node represents a region on a unique call path and stores the profiling data for this call path, such as visits, execution time or hardware events. Score-P provides a special API to read this profile format. In addition to accessing the data per node, it provides access across system entities, such as nodes of the system, processes or threads.

`scorep-autofilter` takes a threshold value (in seconds). All regions with a finer granularity will be filtered. This threshold specification is an optional argument. If no threshold value is provided, the default value is set to 1 ms.

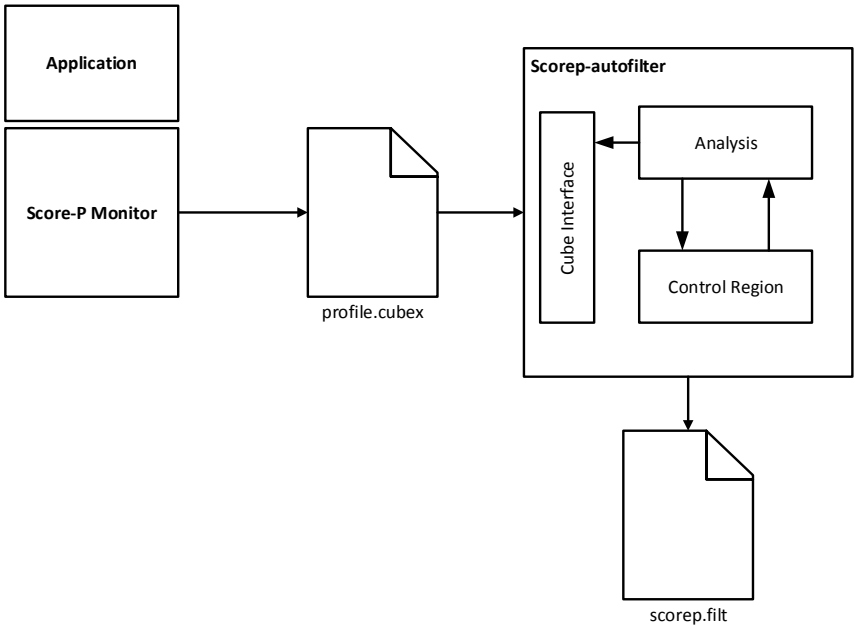

Figure 2: `scorep-auto-filter` architecture.

The granularity of a region is defined as the average execution time of its instances as below:

$$granularity_{reg} = \frac{t_{incl}^{region}}{instances_{excl}} \quad (1)$$

t_{incl} and $instances_{excl}$ are the inclusive execution time and the number of instances of only this region *reg*, not counting instances of nested regions and therefore called *exclusive*.

The tool starts to iterate over all the call path nodes for the instrumented regions. For each call path node, it computes the granularity, which is defined by Equation (1). If $granularity_{reg}$ is lower than the threshold, the tool checks the region type of that call path node in the next step. This is because Score-P can filter only user functions. The name of the function is then added to the Score-P filter file.

Figure 2 shows the architecture of `scorep-autofilter`. The application is instrumented with Score-P and the monitor generates the `profile.cubex` file which is then input to `scorep-autofilter`. The tool itself consists of an *Analysis* component which accesses the profile data through the cube interface. The *Control Region* component determines the region type and outputs the region names into the `scorep.filt` file.

We now illustrate the application of `scorep-autofilter` in the context of the BT-MZ application of the NAS Parallel Benchmarks (NPB) [2]. We first instrument BT-MZ with Score-P

and run the application. A part of the flat profile written into `profile.cubex` is shown in Table 1. The six user functions shown have a granularity below 0.56 μs. The measurement overhead is significant for those regions. The total measurement overhead prolongates one iteration of the progress loop from 0.99 seconds to 7.05 seconds.

Table 1: Performance result of an unfiltered instrumented run of NPB BT-MZ, CLASS=C, NPROCS=4, 10 iterations.

type	visits	time [s]	time [%]	time/visits [μs]	region
USR	115,489,792	50.77	13.8	0.44	matmul_sub
USR	115,489,792	47.20	12.8	0.41	matvec_sub
USR	115,489,792	60.18	16.3	0.52	binvcrhs
USR	68,892,672	25.76	7.0	0.37	exact_solution
USR	4,787,200	2.67	0.7	0.56	lhsinit
USR	4,787,200	2.02	0.5	0.42	binvrhs

Then, we apply `scorep-autofilter` to the generated `profile.cubex` file as shown here:

```
scorep-autofilter -t 0.0001 profile.cubex
```

The threshold value of 0.1 ms is specified by the program argument `-t`. The `-h` option generates help information on the usage of the tool. The tool creates the following Score-P filter file.

```
SCOREP_REGION_NAMES_BEGIN
EXCLUDE
add*
binvcrhs*
binvrhs*
copy_x_face*
copy_y_face*
...
SCOREP_REGION_NAMES_END
```

The `EXCLUDE` section is a rule of the filter file which denotes to exclude the subsequent region names from Score-P measurements. The remaining regions will be considered as candidates for significant regions which is explained in the next section.

After creating the filter file, another run of the application should be executed to verify that the instrumentation overhead was significantly reduced. In this specific example, the execution time of a single iteration of the progress loop in BT-MZ dropped from 7.05 seconds to 1.39 seconds.

2.2 Significant region detection

The second step of DTA determines the significant regions, i.e., regions for which DTA will determine best configurations and for which the RRL will dynamically switch the configuration.

There are three requirements for regions to become significant regions:

1. The region must be coarse enough so that the reconfiguration overhead is negligible.
2. Significant regions cannot be nested. This is required to make the effect of dynamic tuning predictable.
3. The selected significant regions should cover as much of the overall execution time as possible.

The significant region detection as well as the dynamism analysis are implemented in the tool `readex-dyn-detect`. It is again based on the profile measurements of Score-P provided in the `cube` format discussed in the previous section.

After eliminating the measurements for too fine granular regions based on Score-P's filtering mechanism, measurements are typically still provided for a large number of regions. From those regions, the significant regions have to be selected. The workflow is shown in Figure 3.

The detection of the significant regions and the application's dynamism depends on the specification of a phase region (see Deliverable D4.1). The name of the phase region has to be specified as an argument to `readex-dyn-detect`. If no phase region is given, the tool terminates.

The tool first removes all regions that are too fine granular in relation to the reconfiguration overhead from the list of candidate regions. The user can define a threshold for the minimum granularity of regions on the command line.

It also removes regions that are not executed as part of the phase region as well as OpenMP or MPI regions. If, in a later phase of the READEX project, tuning plugins are available for OpenMP or MPI regions, this decision might be reconsidered. The result of this step is a list of candidate regions from which the significant regions are selected.

The next step of the work flow is to run an algorithm which selects the significant regions taking the other two requirements given above, namely that significant regions cannot be nested and should cover most of the execution time, into account. Listing 1 shows the algorithm for selecting significant regions which takes the candidate list as input. The list `sig_list` stores the found significant regions as output.

The algorithm selects only regions that are not dynamically nested, i.e., the execution of a region will not include an execution of any other significant region. For this purpose, the transitive closure of the call graph of the application is constructed and stored in the form of the adjacency matrix `adj_mat`.

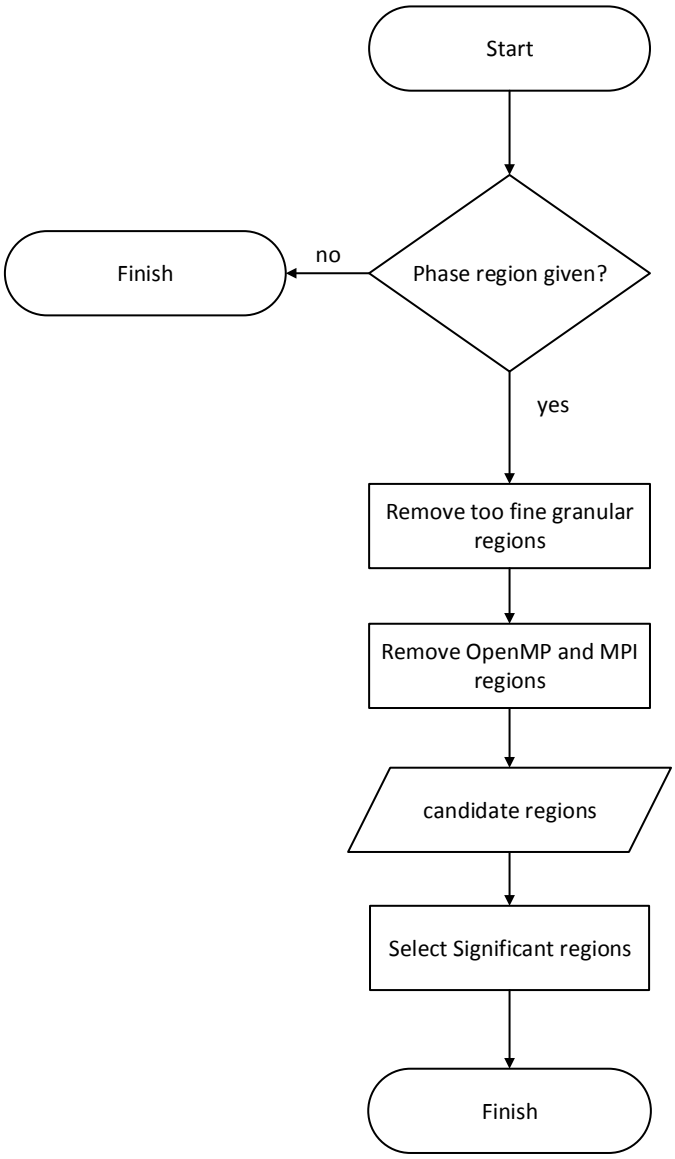

Figure 3: Workflow for the selection of significant regions.

Listing 1: The selection algorithm for significant regions

```
function select_significant_regions(candidate_list) {
   sig_list := empty // list of significant regions
   adj_mat := create_adj_mat_transitive_closure(candidate_list)
   comp_list := get_strongly_connected_comp(candidate_list)
   while (component := comp_list.next())
      if (component has more than one region or is recursive)
          candidate_list := candidate_list - component
      end if
   end while

   while (candidate_list is not empty)
       leaf_region_list := get_leaf_regions(candidate_list)
       while (leaf := leaf_region_list.next())
           parents := leaf.get_immediate_parents()
           Exec_parents^excl := exclusive time of all parents
           if (Exec_leaf^incl>Exec_parents^excl)
                sig_list += leaf
           end if
           candidate_list := candidate_list - leaf;
       end while
       all_parents := sig_list.get_all_parents()
       candidate_list := candidate_list - all_parents
   end while
}
```

The selection algorithm first removes the strongly connected components of the call graph. These are cycles and none of the regions in a cycle can be selected as significant region.

Then, it iterates over the `candidate_list` until the list is empty. In each iteration, it iterates over the leaf regions of the current list of candidates and decides whether the leaf will be selected as significant region. The leaf will be selected if its execution time is more than the summed up exclusive execution times of its parents. Otherwise, it is better to select the parents because they cover more of the execution time.

After all the leafs were processed and removed from the candidate list, also all the predecessors of significant regions are removed from the candidate list as well. These cannot be selected as significant regions, because already selected regions are nested.

Figure 4 shows a call graph as an example for the selection of significant region. (a) presents the original call graph for the candidate regions. In the first step, the strongly connected components are removed. The result is shown in (b). Nodes E and G form a cycle and are removed. Thus, now there are two leaf nodes F and H. Let's assume that the inclusive time

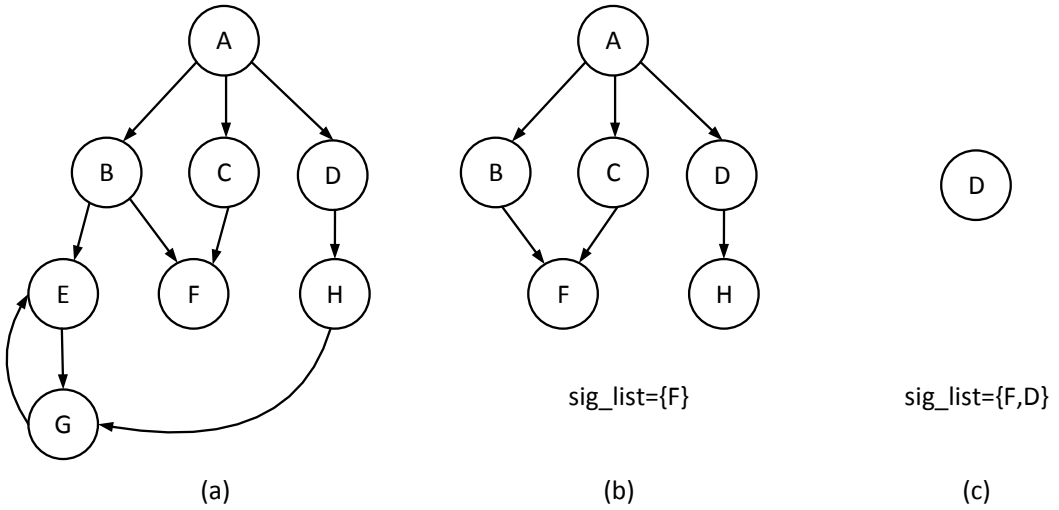

Figure 4: Significant region detection for an example call graph. (a) call graph of candidate regions. (b) Call graph after removal of strongly connected components. (c) Call graph after removal of significant region F and its parents as well as the non-significant leaf region H.

of F is larger than the summed up exclusive times of B and C. Therefore, the algorithm selects F as significant region. For the second leaf node, H, we assume that its inclusive time is less than the exclusive time of D. Therefore, it is not selected as significant region. In (c) we present the status after the leafs and the parents of the significant regions were removed. The final node is D which is now a leaf node and will be selected as significant region as well.

`readex-dyn-detect` takes three arguments related to the detection of significant regions. These arguments are `-t granularity`, `-p phase_region`, and the `profile.cubex` file. The first is the granularity threshold required for reconfiguration, the second the name of the phase region, and the last one the input profile.

The structure of a phase in the BT-MZ benchmark is presented in Figure 5. When `readex-dyn-detect` is applied with the following command line,

```
readex-dyn-detect -t 0.001 -p ProgressLoop profile.cubex
```

it determines the following significant regions: `exch_qbc`, `x_solve`, `y_solve` and `z_solve`. The function `add` was already filtered since its granularity was too fine for the instrumentation. Function `compute_rhs` was not selected since it had a granularity below the given threshold. The tool selected `x_solve`, `y_solve`, and `z_solve` instead of `adi` since most of the compute time was spent in these three routines and more significant regions lead to potentially finer configuration selection for DTA and RRL.

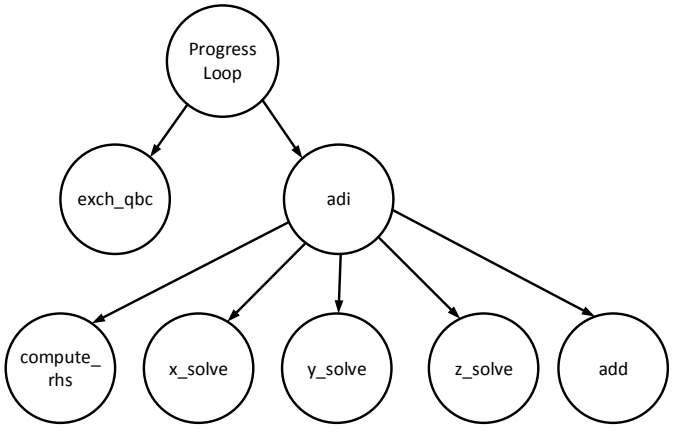

Figure 5: Call graph of NPB BT-MZ.

2.3 Dynamism analysis

After detecting the application's significant region, `readex-dyn-detect` computes the intra- and inter-phase dynamism. Dynamism, i.e., dynamically changing characteristics of the application, is the requirement for successfully tuning applications with the READEX methodology. The tool currently focuses on execution time and compute intensity as main characteristics. Variation in the execution time of significant regions across rts' indicates intra-phase dynamism. Variation in the execution time across rts' of the phase region indicates inter-phase dynamism. Furthermore, different compute intensity, i.e., compute vs. memory bound, of different significant regions also indicates intra-phase dynamism.

The tool can be extended in later project phases if additional tuning relevant characteristics are identified.

To quantify such dynamism and tuning potential, it is essential to measure the performance metrics with statistical information in Score-P. Statistical information is provided by Score-P by requesting the *cube-tuple* profiling format via the `SCOREP_PROFILING_FORMAT` environment variable. This extended cube format provides the number of samples, minimum, maximum, average, and deviation instead of a single absolute value for each metric.

The detection of dynamism can be divided into intra-phase dynamism detection and inter-phase dynamism detection.

2.3.1 Intra-phase dynamism

To detect intra-phase dynamism, `readex-dyn-detect` requires that statistical information is available for the measured metrics via the cube-tuple format. If statistical data are not available, the tool exits and prints a message requesting statistical measurements. To take also the compute intensity of significant regions into account, the total number of instructions

and L3 (LLC) cache misses need to be measured via PAPI. PAPI measurements are triggered by the following specification:

```
export SCOREP_METRIC_PAPI=PAPI_TOT_INS,PAPI_L3_TCM
```

The tool then analyzes for each significant region the variation in the time characteristics. It computes the standard deviation relative to the mean execution time of the region in percent ($deviation_r$) and relative to the mean execution time of the phase ($deviation_p$) as described in Equations (2) and (3) below, respectively. The values characterize how significant the variation in the execution time is for the region and phase execution respectively.

$$deviation_r^{reg} = \frac{dev\_t_{incl}^{reg}}{mean\_t_{incl}^{reg}} * 100 \tag{2}$$

$$deviation_p^{reg} = \frac{dev\_t_{incl}^{reg}}{mean\_t_{incl}^{phase}} * 100 \tag{3}$$

The variation is considered significant if it is beyond a threshold v_t. To decide whether this leads to significant dynamism, the tool computes the computational weight of the region, i.e., its percentage on the phase execution time, according to the formula.

$$weight = \frac{t_{incl}^{reg}}{t_{incl}^{phase}} * 100. \tag{4}$$

If the region's time variation is significant and its weight is larger than a threshold v_w then the tool will report intra-phase dynamism due to that significant region.

Another source of intra-phase dynamism is the variation of other characteristics across different significant regions. The tool currently supports a comparison based on the *compute intensity* of significant regions. It is based on the number of total retired instructions and the number of L3 cache misses. The first one is used as a measure for the work done, the second one for the amount of data transferred between memory and the L3 cache. It should be noted that this measure of transferred data is of course very coarse, since it does not take into account all writes to memory as well as for example hardware prefetching.

The more common definition of compute intensity is based on the floating point instructions, but these cannot be counted on Intel Haswell.

The tool checks all significant regions with a weight above v_w. It compares the compute intensity of those regions and reports intra-phase dynamism if the variation across phase is larger than a threshold v_i.

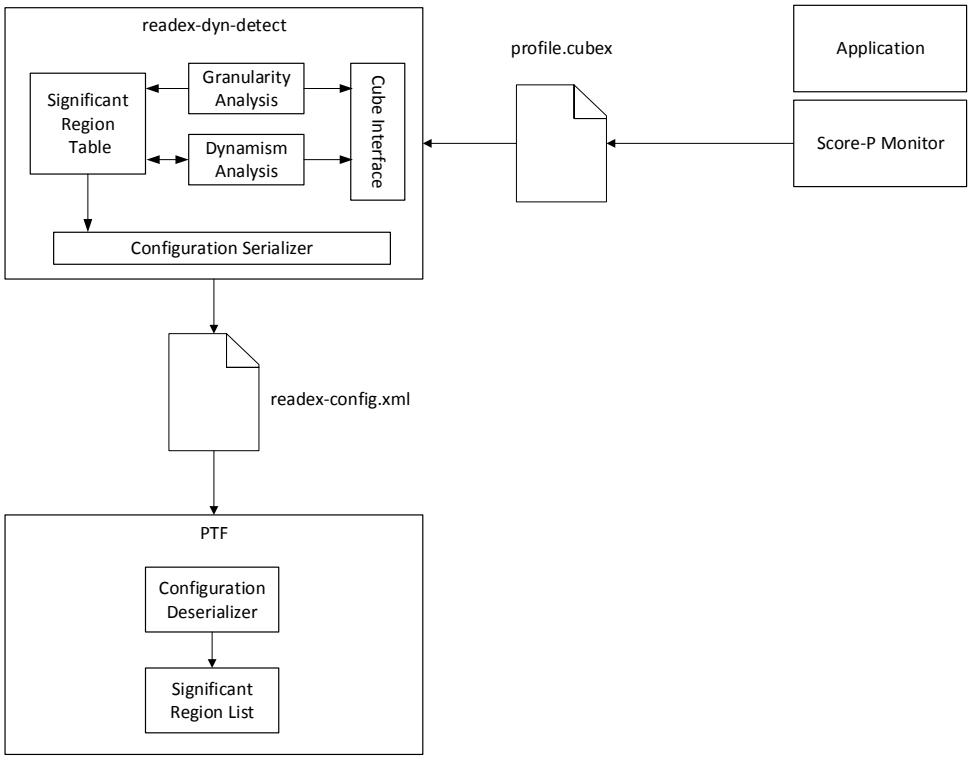

Figure 6: Architecture of `readex-dyn-detect`.

2.3.2 Inter-phase dynamism

Finally, `readex-dyn-detect` investigates whether there is significant inter-phase dynamism in the application. This analysis is based on the execution time of the phase region. It requires the statistical data provided in the cube-tuple format.

The tool computes the relative deviation of the execution times of the phase region $deviation_p^{phase}$. If this deviation is larger than v_t, the tool reports inter-phase dynamism.

2.3.3 The `readex-dyn-detect` architecture

The architecture of `readex-dyn-detect` is given in Figure 6. After the second application run of the improved (with `scorep.filt`) Score-P instrumented application, the performance data with the statistical data are generated in cube-tuple format. This file is then fed into the `readex-dyn-detect` tool to detect the significant regions and to analyze the tuning potential.

The tool accesses the data via the CUBE interface. The *Granularity Analysis* component detects significant regions and inserts them into the *Significant Region Table*. The *Dynamism Analysis* is responsible to determine intra- and inter-phase dynamism. It therefore accesses

the Significant Region Table and the profile data. After the detection of the tuning potential, the list of significant regions is output into an XML configuration file. This file stores the significant regions with the dynamism information as well as the decision of the tool whether significant dynamism exists.

This configuration file is then forwarded to PTF for the DTA. It is deserialized in PTF and the significant region information is translated into a list of PTF region objects. This list is then available to tuning plugins to determine optimized system configurations.

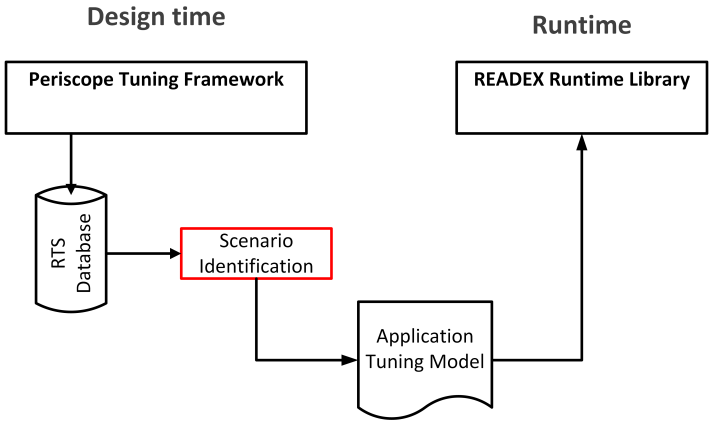

Figure 7: Placement of Scenario Identification in the overall READEX tuning flow.

3 Approach for Scenario Identification

In the final stage of the Design Time Analysis (DTA), runtime situations (rts's) with similar or identical best found configurations are identified before they are clustered into scenarios. Such a clustering limits the number of scenarios and thus reduces the associated runtime overheads.

When the scenarios are generated, a classifier is created so that the observed rts's can be mapped onto their respective scenario. Moreover, a configuration selector is generated for each scenario. The runtime system uses the configuration selector to select the best configuration for a given user objective. Information about the set of scenarios, the classifier and selectors, determined at design time is stored in the Application Tuning Model (ATM) as a serialised in-memory representation. The ATM is read and deserialised at runtime to enable scenario detection and switching. Figure 7 shows the placement of the Scenario Identification module in the overall READEX tuning flow.

The following sections describe the implementation of scenarios, classifiers and configuration selectors in detail.

3.1 Linked Runtime Situations

One of the main pillars of DTA is to use the extensive tuning possibilities of PTF to find the best possible system configuration for each rts. In Section 2.1.4 of Deliverable 4.1 an rts is defined as an instance of a significant region during execution [3]. This definition is very fine-grained, leading to the generation of a very large number of rts's. For example, consider individual executions of a significant region with identical identifier values within a nested loop. This will result in a number of rts's equal to the number of loop iterations. For all practical purposes these rts's are identical, and can therefore be handled together.

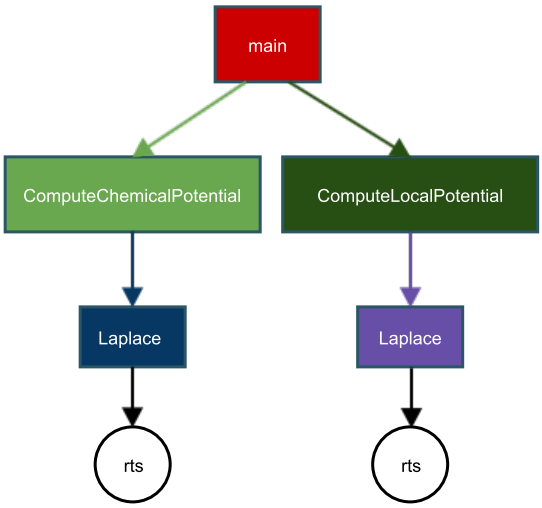

Figure 8: The call tree of a sample application. Only the salient part of the tree is shown.

In order to reduce the number of rts's to be handled individually, we introduce the concept of *linked rts's*. A linked rts is the set of multiple executions of the same significant region with identical context elements. In the set of all rts's, denoted RTS, a linked rts is thus seen as a single rts. For the sake of clarity, we continue to refer to these rts's using the same name and abbreviation as before. Wherever rts is used in this document it is thus to be understood as a linked rts. Due to the coarser granularity of linked rts's, fewer rts's are created. Ultimately, this means that we can group rts's more quickly and create a more compact application tuning model.

3.2 Tuning Results Data Structure

Once PTF has completed its search for the best system configuration for a given rts, the result is stored by an rts object in an RTS Database. When all the rts's have been stored in the RTS database, a scenario identification module (see Figure 7) will retrieve the data stored from this database to group rts's into distinct scenarios. This is possible because the rts's stored in the database contain context element information in addition to the best found configurations. Other information such as measured objective values, and as part of future work, Pareto optimal configurations with corresponding objective values are also stored. The number of occurrences of a given rts, that is, the number of times it is encountered during profiling, can also be included. This information is important for advanced scenario generation, which will be developed later in the READEX project. Figure 8 displays the application call tree of a sample application where the insignificant regions `ComputeChemicalPotential` and `ComputeLocalPotential` both call the significant `Laplace` compute function. Although both of the insignificant regions call the same `Laplace` function, their callpaths differ. Hence,

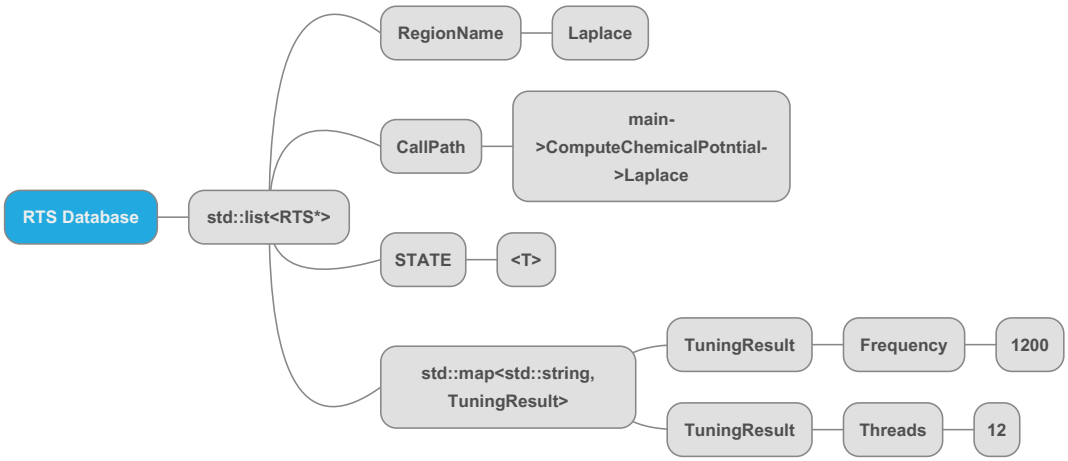

Figure 9: An expanded view of an example rts. The best found system configuration for this particular rts is when the tuning parameter, CPU_Freq is set to 1.2 GHz and the number of threads is 12.

the two calls are considered as two independent rts's and PTF will search for the best configuration for each one and store the results in the RTS Database.

Figure 9 gives an expanded view of example data that can be stored by an rts object in the RTS Database. The identifiers (RegionName, Callpath, and STATE) with their assigned values represents the context elements. A map data structure is used to map a system configuration onto a tuning result. In this simple example, only two tuning parameters are stored, the CPU frequency and the number of CPU threads but an arbitrary number of tuning parameters can be stored. Deliverable D4.1 describes 17 different tuning parameters [3], which may be included in future versions.

3.3 Grouping of rts's into Scenarios

A scenario is a collection of rts's with the same or similar best found configuration. In this section we describe grouping of rts's with identical best found configuration, before we discuss how rts's with similar best found configurations are handled. The notion of *similarity* will be specified together with the description of the grouping.

3.3.1 Grouping rts's with Identical Configuration

In order to detect identical system configurations and cluster the rts's into scenarios, we iterate over the rts's objects from the RTS Database. A new scenario is then created using the system configuration from the rts object, if and only if a scenario with an identical tuning parameter does not already exist. Upon the instantiation of a scenario object, the scenario is assigned a unique id. In parallel, the classifier map data structure described in Section 3.4

is generated linking rts context elements with scenarios. Should a scenario with an identical system configuration exist, the corresponding scenario object is used and the classifier is updated with a link from the new rts to this existing scenario id.

3.3.2 Grouping rts's with Similar Configuration

To illustrate the grouping of rts's with similar configurations, let us consider the example of two rts's that have best found clock frequency settings of 1.2 GHz and 1.3 GHz, respectively. Although these values are different, they may be close to each other with respect to energy consumption or another user-defined objective function. For example, two frequencies are close if the penalty in energy consumption for an rts is small if 1.2 GHz is used instead of its best found frequency of 1.3 GHz. Thus, instead of treating the two rts's with best found clock frequency of 1.2 GHz and 1.3 GHz as different scenarios, which may lead to increased scenario detection and switching overhead at runtime, we group them together in the same scenario.

Clustering rts's with identical best found configurations is rather straightforward. Unfortunately, the same approach cannot be taken with respect to grouping rts's with similar configurations. When grouping rts's with similar configurations, we are looking for configurations that are within the vicinity of each other. This can be challenging since we have identified 17 different system parameters [3], each potentially having their own definition of what its vicinity is.

Finding rts's with similar configuration requires that rts's with identical configurations have already been grouped so that we are only left with rts's that are not identical. An already generated group of rts's with identical configurations are here handled as an individual ungrouped rts. The motivation for the additional grouping of rts's with similar configuration is to reduce switching overheads. In order to group rts's with similar configuration, we compute a similarity score, S_c, which is later used together with a predefined grading scale to effectively group rts's with a similar score.

One important aspect of the similarity score computation is that it is directly associated with the user-defined objective function. This indicates that if the user decides to tune his or her application for performance or energy consumption, the weight of the individual scores are changed accordingly. For example, in situations where the objective function is to tune for performance, tuning parameters that influence the performance are given a higher score, as opposed to tuning parameters that do not impact the performance as much, which are thus given a lower score.

In order to compute the similarity score for an rts, we iterate over the tuning configuration using the following formula:

$$S_c = \sum_{i=0}^{tp} w_i \cdot s_i \tag{5}$$

where w is a weight score associated with the tuning parameter type, s is the score associated with the tuning parameter value and tp is the number of tuning parameters. Once the similarity score has been computed for each rts, it is used for comparison across other ungrouped rts's. The weights and score ranges for different tuning parameters will be defined based on expert designer knowledge, combined with experiments performed on a given platform using micro benchmarks. This will reveal both the cost of switching a parameter from one value to another, and the expected influence of a given parameter on the total objective value. Later in the READEX project we will also investigate the possibility of using the results of multiple PTF tuning experiments to reveal the penalty of using a suboptimal parameter value, for example a clock frequency of 1.2 GHz instead of the best found 1.3 GHz.

As an illustrative example, we assume that we have three tuning parameters, CPU core clock frequency (F), the number of OpenMP threads (T) and the uncore frequency (U). Moreover, we also assume that the user-defined objective function is to tune for maximum performance. Based on the given objective function, we start by weighting the various tuning parameters types, where for example $F = 0.7$, $T = 0.2$, $U = 0.1$. Next, we iterate over the different tuning parameter values, according to (5), and grade them based on their value on a scale from one to ten, where one is the lowest score and ten is the best score. We prefix the score scale for tuning parameter values that are range-based, such as clock frequency. In this particular example, the highest clock frequency would be given a score of ten, while the lowest clock frequency would be given a score of one. Similarly, the same score would be applied to the number of OpenMP threads and to the uncore frequency. This way, the similarity score of two rts's may be close to each other even if some tuning parameters are quite different, as long as the tuning parameters with high weight are close. Iterating over all rts's, those that are close, according to the predefined scale, are grouped into the same scenario.

The same configuration will be used for each resulting scenario. This results in an overhead for those rts's that could have used an even better configuration. On the other hand, too many scenarios result in a more complex ATM and more frequent switching between configurations, which also has an overhead. Furthermore, the overhead of grouping two rts's is not only related to how similar their configurations are. For instance, if an rts only occurs once during the execution of an application, the overall overhead of using a suboptimal configuration is insignificant. These types of trade-offs are handled by the similarity score algorithm and particularly the score range that can be extended for rts's that occur seldom. They are also taken into account when the common configuration is selected for a given scenario.

As part of the runtime calibration mechanism, scenarios generated as described in this subsection may be split and regrouped if the measured objective values do not correspond to the DTA measured values.

3.4 Scenario Classifier Generation

The scenario classifier is used at runtime to determine the upcoming scenario based on the current context elements (identifier and their values). We implement a map data structure where concatenations of rts identifier values are keyed to their respective scenario ids. For

example, if the callpath is the only identifier, it is used as a key. The map data structure is built and populated during DTA. Once the map has been successfully built, an in-memory representation of the map is saved to disk for later use at runtime. The Cereal framework [1] is used for the serialisation.

At runtime, the map is deserialised by the Tuning Model Manager (TMM), which is a module of the READEX Runtime Library (RRL) that acts as a mediator between DTA and the Runtime Application Tuning (RAT). Figure 7 shows how the ATM is used to connect and transfer information between DTA and RAT.

3.5 Configuration Selector Generation

The configuration selector is used at runtime to select which configuration to use, that is, which tuning parameter settings to apply. This will be based on the upcoming scenario determined by the classifier. If the scenario consists of rts's with identical configurations, these parameter settings are used directly. If the scenario holds rts's with similar configurations, parameter settings are selected at design time according to the principles outlined in Section 3.3.2.

In its simplest form the selector is implemented as a function that returns the selected scenario's system configuration and other associated values based on parameters that are passed to the function. This is possible because the selector maps a scenario id onto a system configuration, which like the classifier is serialised during the latter stages of DTA.

Later in the READEX project more advanced configuration selector generation mechanisms will be developed. Evaluation of switching overhead will be considered as well as Pareto optimal configurations found for a given scenario. This can require individual selectors for each scenario.

4 Summary

In this deliverable, we have presented the READEX tools developed for tuning potential analysis and for scenario identification. Both are important parts of the Design Time Analysis stage, its first and last step, respectively. We have described how the two automatic tools `scorep-autofilter` and `readex-dyn-detect` are used for automatic reduction of instrumentation overhead, for significant region detection, and dynamism analysis. The final result of this process is first and foremost a conclusion regarding whether sufficient dynamism is present in the application for READEX to exploit. If so, the step also outputs a set of significant regions that are coarse grained enough for reconfiguration overhead to be negligible, that are not nested in other significant regions, and that cover as much as possible of the overall execution time.

When the tuning potential analysis is finished the Periscope Tuning Framework is used to search for the best possible system configuration for each runtime situation (rts). An rts is here understood as the set of multiple executions of the same significant region with identical context elements (identifiers with values). The result of this search is stored in an RTS Database used as input to the scenario identification. In this deliverable we have described how rts's are grouped into scenarios according to similarity of their best found system configurations. Furthermore, we have described how scenario classifiers and configuration selectors are generated and how they, together with the scenario set, are stored in the Application Tuning Model as a serialised in-memory representation. The ATM is later read and deserialised at runtime to enable scenario detection and configuration switching. In the deliverable we have described how the grouping of rts's in scenarios, and use of efficient classifiers and selectors reduce the scenario detection and switching overhead at runtime.

References

[1] cereal - a c++11 library for serialization. `http://uscilab.github.io/cereal/index.html`. Last accessed August 25, 2016.

[2] David H. Bailey. The NAS Parallel Benchmarks. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1254–1259. Springer, New York, 2011.

[3] Michael Lysaght, Kashif Iqbal, Joseph Schuchart, Andreas Gocht, Michael Gerndt, Anamika Chowdhury, Madhura Kumaraswamy, Per Gunnar Kjeldsberg, Magnus Jahre, Mohammed Sourouri, David Horák, Lubomír Říha, Radim Sojka, Jakub Kruzik, Kai Diethelm, and Othman Bouizi. D4.1: Concepts for the READEX tool suite. *READEX WP4 Report*, 2016.

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